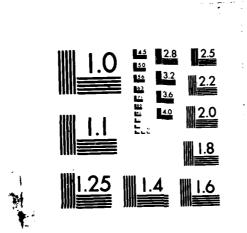
AD-A138 891 DISTRIBUTED DATABASE CONTROL AND ALLOCATION VOLUME 1 FRAMEWORKS FOR UNDER. (U) COMPUTER CORP OF AMERICA CAMBRIDGE MA H K LIN ET AL OCT 83 RADC-TR-83-226-VOL-1 F30602-81-C-0028 F/G 9/2 1/4 % UNCLASSIFIED NL



AND STATES OF STATES STATES STATES STATES STATES STATES

MICROCOPY RESOLUTION TEST CHART NATIONAL BUREAU OF STANDARDS-1963-A

RADC-TR-83-226, Vol I (of three) Final Technical Report October 1983

AD A138891



DISTRIBUTED DATABASE CONTROL AND ALLOCATION Frameworks for Understanding Concurrency Control and Recovery Algorithms

Computer Corporation of America

Wente K. Lin, Philip A. Bernstein, Nathan Goodman and Jerry Nolte

APPROVED FOR PUBLIC RELEASE; DISTRIBUTION UNLIMITED

THE FILE CORY

ROME AIR DEVELOPMENT CENTER Air Force Systems Command Griffiss Air Force Base, NY 13441



84 03 12 005

This report has been reviewed by the RADC Public Affairs Office (PA) and is releasable to the National Technical Information Service (NTIS). At NTIS it will be releasable to the general public, including foreign nations.

RADC-TR-83-226, Vol I (of three) has been reviewed and is approved for publication.

APPROVED:

Emilie J. Siarkiewicz

EMILIE J. SIARKIEWICZ Project Engineer

APPROVED:

JOHN J. MARCINIAK, Colonel, USAF Chief, Command and Control Division

FOR THE COMMANDER:

DONALD A. BRANTINGHAM Plans Office

al A Browling

If your address has changed or if you wish to be removed from the RADC mailing list, or if the addressee is no longer employed by your organization, please notify RADC (COTD) Griffiss AFB NY 13441. This will assist us in maintaining a current mailing list.

Do not return copies of this report unless contractual obligations or notices on a specific document requires that it be returned.

UNCLASSIFIED

SECURITY CLASSIFICATION OF THIS PAGE (When Date Entered)

REPORT DOCUMENTATION PAGE	READ INSTRUCTIONS BEFORE COMPLETING FORM	
	3. RECIPIENT'S CATALOG NUMBER	
RADC-TR-83-226, Vol I (of three) AD-A1388		
4. TITLE (and Subtitle)	5. TYPE OF REPORT & PERIOD COVERED	
DISTRIBUTED DATABASE CONTROL AND ALLOCATION Frameworks for Understanding Concurrency	Final Technical Report Jan 1981 - Jan 1983	
Control and Recovery Algorithms	6. PERFORMING ORG. REPORT NUMBER	
	N/A	
7. AUTHOR(a)	S. CONTRACT OR GRANT NUMBER(s)	
Wente K. Lin Nathan Goodman	F30602-81-C-0028	
Philip A. Bernstein Jerry Nolte		
9. PERFORMING ORGANIZATION NAME AND ADDRESS	10. PROGRAM ELEMENT, PROJECT, TASK AREA & WORK UNIT NUMBERS	
Computer Corporation of America		
Four Cambridge Center Cambridge MA 02142	62702F 55812121	
11. CONTROLLING OFFICE NAME AND ADDRESS	12. REPORT DATE	
	October 1983	
Rome Air Development Center (COTD)	13. NUMBER OF PAGES	
Griffiss AFB NY 13441	300	
14. MONITORING AGENCY NAME & ADDRESS(If different from Controlling Office)	15. SECURITY CLASS. (of this report)	
	UNCLASSIFIED	
Same	154. DECLASSIFICATION/DOWNGRADING	
	N/A SCHEDULE	
16. DISTRIBUTION STATEMENT (of this Report)		
Approved for public release; distribution unlimited		
17. DISTRIBUTION STATEMENT (of the abstract entered in Block 20, If different from	om Report)	
Same		
18. SUPPLEMENTARY NOTES		
RADC Project Engineer: Emilie J. Siarkiewicz (COTD)		
19. KEY WORDS (Continue on reverse side if necessary and identify by block number)		
Distributed Databases		
Concurrency Control		
Reliability		
20. ABSTRACT (Continue on reverse side if necessary and identify by block number)		
This is the first of three volumes of the final technical report for the		
project "Distributed Database Control and Allocation". This volume des-		
cribes frameworks for understanding concurrency control and recovery		
algorithms for centralized and distributed database systems. The second		
volume describes work on the performance analysis of concurrency control algorithms. The third volume summarizes the results in the form of a		
distributed database designer's handbook.		
The state of the s		
AA PURE 4.40A		

DD FORM 1473 EDITION OF I NOV 68 IS OBSOLETE

UNCLASSIFIED

UNCLASSFIEID

SECURITY CLASSIFICATION OF THIS PAGE(When Date Entered)

This volume is an anthology of papers organized in two parts. Part One covers concurrency control and consists of four papers. Part Two covers recovery algorithms and consists of four papers.

The first paper presents an overview of concurrency control algorithms for distributed database systems. It decomposes the concurrency control problem into several subproblems, and describes the known solutions to the subproblems. This paper extends Bernstein and Goodman's earlier survey of the subject ("Concurrency Control in Distributed Database Systems," ACM Computing Surveys 13,2 (June 1981) by using an improved decomposition into subproblems and by including more algorithms, notably certifiers and multiversion algorithms.

In a multiversion concurrency control algorithm, each write operation on a data item, x, produces a new "version" of x, leaving old versions of x unchanged. The second paper presents a comprehensive description and logical analysis of multiversion concurrency control algorithms. It extends serial izability theory to handle the multiversion semantics of "write." It describes multiversion concurrency control algorithms based on locking and timestamping, and proves them correct using the extended theory.

The third and fourth papers present mathematical analyses of two-phase locking algorithms. The third paper describes a queueing theoretic approach coupled with a random graph analysis of deadlocks. The fourth paper describes a new control theoretic analysis that uses significantly weaker assumptions than the standard queueing theoretic approach; thus, its conclusions are quite general. These analyses only study a few of the many available concurrency control algorithms, and therefore are not comprehensive. They principally demonstrate the feasibility of these approaches, but leave open a more complete comparative analysis of algorithms. Such a comparative analysis using simulation techniques appears in the second volume of this report.

A nonmathematical survey of these algorithms is presented in the fifth paper. It describes methods for: undoing a transaction after it aborts; redoing committed transactions after a site fails; extending any centralized undo or redo algorithm to a distributed system.

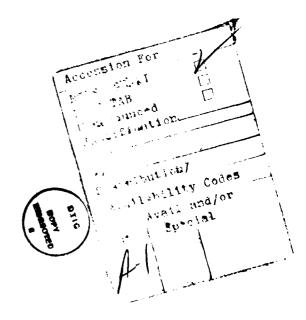
A mathematical model of recovery algorithms is presented in the sixth paper. Each of the algorithms described in the previous paper is proved correct using this model.

The seventh paper presents a new algorithm for site recovery in a distributed database system. The problem is how to bring a site's database up to date after it has recovered from failure. The solution allows different portions of the database to be brought up to date independently, thereby avoiding a strongly synchronized bulk transfer of the entire database.

The last paper presents an optimal algorithm for undoing a set of transactions to a consistent state. The problem is that when one transaction is undone, any transaction that read its output must also be undone. Thus, undo's may cascade requiring many other transactions to be undone. It is shown that there is a unique best set of transactions to undo, and a fast algorithm for finding this set is described.

CONTENTS

		Page
	Executive Summary	iii
I.	Introduction	1
II.	A Sophisticate's Introduction to Distributed Database Concurrency Control	4
III.	Multiversion Concurrency Control Theory and Algorithms	52
IV.	Performance Analysis of Concurrency Control Methods in Database Systems	90
v.	A Simple Analytic Model for Performance of Exclusive Locking in Database Systems	118
VI.	Recovery Algorithms for Database Systems	172
VII.	An Operational Model for Database System Reliability	1 98
ZIII.	Site Initialization, Recovery, and Back-Up in a Distributed Database System	255
ıx.	An Algorithm for Minimizing Roll Back Cost	273



Distributed Database Control and Allocation

Executive Summary

The design of a distributed database management system (DDBMS) involves many critical design decisions. It is recognized that one of the most important of these design decisions is the choice of the concurrency control algorithm to be used. Many concurrency control algorithms for DDBMSs have been proposed (48 principal ones were identified in the previous effort*), but few studies have been undertaken to rigorously compare their performance and other characteristics. One possible reason for this is that, in detail, these algorithms seem very different, thus making comparison difficult. As a result, the DDBMS designer finds it difficult to choose the concurrency control algorithm which is given the design parameters of the particular system under consideration.

Consequently, the Distributed Database (DDB) Control and Allocation project had the following objectives:

- 1. Review the distributed concurrency control research published in the literature and incorporate that research into the taxonomy of the distributed database concurrency control algorithms. Based on this taxonomy, develop a new framework for distributed database control.
- 2. Develop new distributed database concurrency control algorithms using the framework developed in 1.
- 3. Simulate the performance of the distributed database concurrency control algorithms that are found to be dominant in the previous study.
- 4. Build an analytical model of distributed database concurrency control.
- 5. Survey the current studies of reliability and recovery of distributed database systems and the analysis of published algorithms.

^{*}This effort was a follow-on to a previous effort conducted by the same research team which is documented in "Fundamental Algorithms for Concurrency Control in Distributed Database Systems," Philip A. Bernstein, et al, RADC TR-80-158, May 1980.

- 6. Develop a framework for reliability and recovery of distributed database systems.
- 7. Consolidate the results of the previous tasks into a system designer's handbook.

The <u>first</u> objective was achieved by means of the framework discussed in Section II of Volume I. The framework facilitates the taxonomy of distributed concurrency control algorithms by identifying the essential component functions of distributed concurrency control mechanisms. The following is a summary of that section.

A distributed database system (DDBS) is a database system that provides commands to read and write data that is stored at multiple sites of a network. If users access a DDBS concurrently, they may interfere with each other by attempting to read and/or write the same data. Concurrency control is the activity of preventing such behavior. The following simple model of DDBS structure and behavior highlights those aspects of a DDBS that are important for understanding concurrency control.

A database consists of a set of data items (e.g., simple variable, file, record, or page, etc.). Users access data items by issuing read and write operations. Read(x) returns the current value of x, while write(x, new-value) updates the current value of x to new-value. Users interact with the DBMS by executing programs called transactions. Each site of a DDBS runs one or more of the following software modules: a transaction manager (TM), a data manager (DM), or a scheduler. Transactions talk to TMs, TMs talk to schedulers, schedulers talk among themselves and to DMs, DMs manage the data. Each transaction issues a 'begin' operation to its TM when it starts executing and an 'end' when it is finished. forwards each read and write to a scheduler. The scheduler controls the order in which DMs process reads and writes. The DM executes each read and write it receives for the data items at its site.

The DDBS modules that are most important to concurrency control are schedulers. There are three types of schedulers: two-phase locking, timestamp ordering, and serialization graph checking. Two-phase locking requires setting a read (or write) -lock for transaction Ti before outputting ri[x] (or wi[x]). The lock must be held at least until the operation is executed by the appropriate DM. Different transactions cannot simultaneously hold 'conflicting' locks. Two locks conflict if they are on the same data item and at least one is a write-lock. In timestamp ordering (T/O) each transaction is assigned a globally unique timestamp by its TM. All pairs of conflicting operations are then output in timestamp order. A serialization graph (SG) is a directed graph whose nodes are transactions, such as TO, ..., Tn, and

whose edges are all Ti-->Tj, such that, for some x, (1) Ti reads x before Tj writes x, or (2) Ti writes x before Tj reads x, or (3) Ti writes x before Tj writes x. A serialization graph checking scheduler works, therefore, by explicitly building a serialization graph and checking it for cycles.

Each of these schedulers can be adapted to work as a certifier (the term 'certifier' refers to a scheduling philosophy rather than a specific scheduling rule). A certifier makes its decisions on a per-transaction basis. That is, when it receives an operation, it internally stores information about the operation and outputs it as soon as all earlier conflicting operations have been acknowledged. When a transaction ends, its TM sends an 'end' operation. At this point, the certifier checks its stored information to see whether the transaction executed serializably. If it did, the certifier certifies the transaction, allowing it to terminate; otherwise, it aborts the transaction.

In addition to the type of scheduler, the location of the scheduler and how replicated data is handled must be into consideration for distributed database concurrency control. Instead of one scheduler for the whole system, there is one scheduler per DM. scheduler normally runs at the same site as the DM and schedules all operations that the DM executes. There are three approaches to handling data replication: nothing', 'primary copy', and 'voting'. In the 'do nothing' approach each read reads from the latest transaction preceding it that wrote into any copy of the data item and writes into all copies of each data item using a serializable scheduler. In the 'primary copy' approach, some copy of each data item is designated the primary copy, such that each TM translates ri[x] into ri[xj] for some copy xj, but translates wi[x] into a single write on the primary copy and the primary copy's scheduler issues writes on the other copies of x. In the 'voting' approach, writes are issued to all copies of each data item and when the scheduler is ready to output wi[xj], it sends a vote of 'yes' to the vote collector for x. When the vote collector receives 'yes' votes from a majority of schedulers, it tells all schedulers to output their writes.

The second objective of developing new algorithms using this framework is achieved by the algorithms described in Section III of Volume I. That section describes the work in extending concurrency control theory to multiversion databases. In a multiversion database each write operation on a data item, x, produces a new "version" of x, leaving old versions of x unchanged. When transactions issue operations that specify data items, the system must translate these into operations that specify versions. In a single version database, concurrency control correctness depends on the order in which reads and writes are processed. In a multiversion database, correctness depends on translation as well

as order.

The main idea is one-copy serializability: an execution of transactions in a multiversion database is one-copy serializable (1-SR) if it is equivalent to a serial execution of the same A multiversion transactions in a single version database. concurrency control algorithm is correct if all of its executions Effective necessary and sufficient conditions for an execution to be 1-SR were derived which use the concept of version A graph structure, multiversion serialization graphs (MVSGs), that helps check these conditions is given. Once a version order is fixed, an execution is 1-SR if and only if its MVSG is acyclic. MVSGs are analogous to the serialization graphs widely used in single version concurrency control theory. The theory was applied to three multiversion concurrency control algorithms. One algorithm uses timestamps, one uses locking, and one combines locking with timestamps.

The third objective of simulating and evaluating the performance of distributed database concurrency control algorithms is documented in Sections II through VI of Volume II. Section II presents a study that analyzed the relationship between the performance of the two-phase locking algorithm and the following system parameters: access distribution of the database, data granularity, transaction size, and multiprogramming level. In a distributed database system, communication delay is also a major factor affecting the performance of a concurrency control algorithm. Section III documents an analysis of the relationship between the performance of the two-phase locking algorithm and communication delay.

Another important factor that affects performance is the number of read-only transactions relative to the number of write transactions, i.e., the ratio of read-only to write transactions. Section IV documents an analysis of the relationship between the performance of the two-phase locking algorithm and that ration.

Section V extends the analysis to algorithms based on timestamps by presenting a comparison of the performance of three distributed concurrency control algorithms — the basic timestamp, multiple version timestamp, and two-phase locking. Section VI documents the analysis of the two-phase locking algorithm in more detail and the refinement of the algorithm into nine algorithms. In addition, the previous two timestamp algorithms were reevaluated in more detail and a new timestamp-based algorithm, the dynamic timestamp algorithm, was analyzed. The performance of the twelve algorithms was then compared. The results of these simulation studies could form the basis for designing a distributed database designer's aid. The designer's aid would help the system designer to design distributed transactions, partition the database into fragments, replicate and distribute the fragments, and choose the concurrency control algorithm that performs best in his system environment.

The <u>fourth objective</u> of analytical modeling of the distributed concurrency control algorithms was achieved through the analytical models described in Sections IV and V of Volume I. Section IV describes using a queueing theoretic approach coupled with a random graph analysis of deadlocks. An analytical model was developed for

the general two-phase locking algorithm, which can be used to estimate the steady state rates of conflicts and deadlocks in the system. Dynamic two-phase locking was analyzed, proceeding from a simple deadlock prevention two-phase locking algorithm (which gives worst case bounds on the performance of any deadlock preventing two-phase locking algorithm) to the general case of two-phase locking where deadlocks are allowed. Under reasonable assumptions for transaction behavior, it was found that the rate of deadlocks is proportional to the average number of transactions in the system and the rate of conflicts is proportional to the mean of the product of the number of free transactions in the system multiplied by the total number of transactions in the system.

Section V describes using a new control theoretic analysis that uses significantly weaker assumptions than the standard queueing theoretic approach, thus, its conclusions are quite general. The model presented is easy to understand and costs little to solve computationally, that captures the essential features of the system The model has two parts: a flow diagram and a set of equations describing the behavior of the system. The equations are derived using the steady state average values of the variables. The underlying idea is to characterize the system in terms of those of average values, instead detailed dynamics involving instantaneous values of each variable.

These analyses only studied a few of the many available concurrency control algorithms, and, therefore, are not comprehensive. They principally demonstrated the feasibility of these approaches.

The survey/study of reliability and recovery of distributed database systems that achieved the <u>fifth objective</u> is documented in Sections VI through IX of Volume I. The following paragraphs contain a brief summary of the current taxonomy of recovery algorithms as well as a description of Sections VI-IX.

The recovery algorithm of a DBS avoids incorrect states caused by transaction failures and system failures by ensuring that the database only includes updates that are produced by transactions that execute to completion. A centralized DBS is modeled as storage, a scheduler, and a recovery system.

The storage component consists of buffer storage and stable storage. Both are divided into physical pages of equal and fixed size. Buffer storage models main memory and stable storage models disk memory. The DB consists of a set of logical pages. A transaction is a program that can read from or write into the DB and can issue four types of commands: read, write, commit, and abort. The scheduler controls the order in which these commands are passed to the recovery system and guarantees that the execution is recoverable. The recovery system processes the read, write, commit, and abort commands it receives from the scheduler and handles system failures.

Recovery algorithms often store copies of pages that were recently written on on an audit trail (sometimes called a

journal or log). For each write processed by the algorithm, the audit trail may contain the identifier of the transaction that performed the write, a copy of the newly written page (called the after-image, and a copy of the physical page in the stable database that was overwritten by the write (called the before-image). Different algorithms vary considerably in the information they keep on the audit trail and in how they structure that information. Recovery algorithms also differ in the time at which they write pages into the stable database. They may perform such writes before, concurrently with, after the atomic instruction that ommits the transaction that last wrote those pages. In a page that is written by an active transaction is wr ten into the stable database before the transaction com ...s and the transaction aborts due to a system c transaction failure, the recovery algorithm must undo . write by restoring the previous copy (before-image _ the page. If a page that is written by an active transaction is not written into the stable database before the transaction commits and a system failure occurs after the transaction commits, but before the page is written into the stable database, the recovery algorithm must redo the write by moving the page to the stable database.

Recovery algorithms can be categorized based on the timing of updates to the stable database. Consequently, there are four basic types of recovery algorithms: ones that require undo but not redo, redo but not undo, both undo and redo, and neither undo nor redo.

As previously stated, a DDBS is modeled by a set of transaction managers (TMs), data managers (DMs), and schedulers. A DM is a centralized DBS as defined above. It processes reads and writes on pages stored at its site. It also processes commits and aborts, which permanently install or undo the writes of a transaction at the DM site. A TM interfaces transactions and DMs. Unfortunately, TMs and DMs may fail at unpredictable times. Each TM must process commands so that failures of other TMs and/or DMs never cause it to produce incorrect results. Consequently, each TM keeps an active transaction list, a commit list, and an abort list in stable storage for reference against TM failures.

To avoid inconsistencies caused by TM failures, there are two basic commit algorithms: two-phase commit and three-phase commit. In two-phase commit a TM does not send 'commit(i)' to any DM until every DM in 'active(i)' has transaction T(i)'s after-images on stable storage. In three-phase commit each TM has one or more backup TMs, such that if a TM fails, the backup can take over its functions. In particular, a three-phase commit is a two-phase commit with an added step: the TM sends a 'precommit(i)' to each backup and waits for all backups to acknowledge before it sends 'commit(i)' to each DM in 'active(i)'. Aborts are handled in a similar way.

To avoid delay caused by the failure of a DM, the DBS can replicate data, that is, store parts of the database at more than one DM site. If one copy is unavailable due to a DM failure, other copies can be used instead. Replication of data, however, introduces another dimension to the consistency problem that involves the concurrency control algorithms described elsewhere in this report.

Section VI of Volume I presents a non-mathematical survey of recovery algorithms and describes methods for undoing a transaction after it aborts, redoing committed transactions after a site fails, and extending any centralized undo or redo algorithm to a distributed system.

Section VII presents a mathematical model of recovery algorithms (which is described below) and the correctness proof using this model for each of the algorithms described in Section VI.

Section VIII describes a new algorithm for site recovery in a distributed database system. The problem is how to bring a site's database up to date after it has recovered from failure. The solution presented allows different portions of the database to be brought up to date independently, thereby avoiding a strongly synchronized bulk transfer of the entire database to the recovered site.

Section IX describes an optimal algorithm for undoing a set of transactions to a consistent state. The problem is that when one transaction is undone, any transaction that read its output must also be undone. Thus, undos may cascade requiring many other transactions to be undone. It is shown that there is a unique best set of transactions to undo, and a fast algorithm for finding that set is described.

Because the subject of reliability and recovery of DDBSs is relatively unexplored, only a few algorithms were reported. Further research is needed to discover new algorithms.

A framework for the reliability and recovery of a distributed database system achieving the <u>sixth objective</u> is described in Section VII of Volume I. The framework consists of an operational, state-based model for studying reliability of DBSs, i.e., the system is described at any point in time by a "system state". Reliability-related properties of the system (e.g., "resiliency") can be expressed as predicates on the system state. Transaction processing algorithms can be described as state transition functions, mapping the current system state to the next. Finally, correctness and other reliability properties of algorithms can be proved formally by examining the system state sequences that can be generated by the algorithms in question.

This framework captures the essential components of existing reliability and recovery algorithms. But, because research on this subject is in its primitive stages, more research is needed to refine the framework and to use it to develop more efficient algorithms. Moreover, the refined framework could become a basis for the standardization of distributed reliability and recovery architectures.

Finally, all these results were summarized in a separate Distributed Database System Designer's Handbook (objective seven), which is included as Volume III. This handbook can help the designer to select a distributed concurrency control algorithm that performs best in his system environment. The following is a summary of those results and recommendations.

The twelve concurrency control algorithms identified in the previous effort as being dominant methods were selected for further study. These algorithms are called

- (1) primary site and primary site two-phase locking,
- (2) primary copy and primary copy two-phase locking,
- (3) basic and basic two-phase locking,
- (4) basic and primary copy two-phase locking,
- (5) basic and primary site two-phase locking,
- (6) DDM multiple version and optimistic two-phase locking (DDM),
- (7) basic and optimistic two-phase locking,
- (8) majority consensus timestamp,
- (9) wait-die two-phase locking,
- (10) basic timestamp,
- (11) multiple version timestamp, and
- (12) dynamic timestamp.

Five of the twelve were found to perform best in various system environments: basic timestamp, multiple version timestamp, DDM, basic-optimistic two-phase locking, and basic-primary two-phase locking.

When most transactions are short, algorithms that abort conflicting transactions (such as basic timestamp multiple version timestamp) perform better than algorithms that block conflicting transactions (such as basic-primary). this environment, transactions Ιn conflict rarely; and when they do conflict, the blocking be longer than the average transactions tend to transaction size and blocking delay. If a two-phase locking algorithm must be used, algorithms that delay lock conflict checking (such as the DDM and basic-optimistic) perform better than those that expedite lock conflict checking (such as basic-primary). the communication bandwidth is very high, communication delay can devastate system performance; thus, designer should reduce communication delay by locally controlling and accessing data as much as possible.

However, no matter which concurrency control algorithm

the designer uses, a system that has long transactions always performs worse than a system that has short transactions. The designer should design transactions to access as much data in parallel as possible, and to break long transactions into shorter transactions.

The performance study showed that no one algorithm performs best in all system and application environments. If the system environment is stable, the database designer can select one algorithm that performs best in the environment. If the system environment is not stable, the database designer can assign different weights to different environments according to how often the system stays in each environment. The database designer then selects the algorithm that has the best weighted average performance.

From the results, it can be concluded that the best algorithm would be one that could be adjusted by the system administrator, according to the environment, to use transaction abortion and delay lock conflict detection whenever transactions are short, and to use transaction blocking and detect lock conflicts as soon as possible whenever transactions are long. The adjustable algorithm would also alternate, depending on the load on the communication channel, between algorithms that have more localized control and algorithms that have more distributed control.

In summary, all of the objectives were satisfactorily met. The next step would be to refine the taxonomy for reliability and recovery algorithms and conduct performance evaluations for existing and new algorithms, as was done in this effort for concurrency control. A successive task would then be to translate the results of both evaluation efforts into a practical, integrated set of tools that aid distributed database designers, and into a standard architecture of distributed DBMS that facilitates the interconnection of different DBMSs.

SECTION I

INTRODUCTION

This is the first volume of the final technical report for the project "Distributed Database Control and Allocation," sponsored by Rome Air Development Center, contract number F30602-81-C0028. This volume describes frameworks for understanding concurrency control and recovery algorithms for centralized and distributed database systems. The second volume describes work on the performance analysis of concurrency control algorithms.

This volume is an anthology of papers organized in two parts.

Part One covers concurrency control and consists of Sections II through

V. Part Two covers recovery and consists of Sections VI through IX.

Section II presents an overview of concurrency control algorithms for distributed database systems. It decomposes the concurrency control problem into several subproblems, and describes the known solutions to the subproblems. This paper extends Bernstein and Goodman's earlier survey of the subject ("Concurrency Control in Distributed Database Systems," ACM Computing Surveys 13,2 (June 1981)) by using an improved decomposition into subproblems and by including more algorithms, notably certifiers and multiversion algorithms.

In a multiversion concurrency control algorithm, each write operation on a data item, x, produces a new "version" of x, leaving old versions of x unchanged. Section III presents a comprehensive description and logical analysis of multiversion concurrency control algorithms. It extends serializability theory to handle the multiversion semantics of "write." It describes multiversion concurrency control algorithms based on locking and timestamping, and proves them correct using the extended theory.

Section IV and V present mathematical analyses of two-phase locking algorithms. Section IV uses a queueing theoretic approach coupled with a

random graph analysis of deadlocks. Section V uses a new control theoretic analysis that uses significantly weaker assumptions than the standard queueing theoretic approach; thus, its conclusions are quite general. These analyses only study a few of the many available concurrency control algorithms, and therefore are not comprehensive. They principally demonstrate the feasibility of these approaches, but leave open a more complete comparative analysis of algorithms. Such a comparative analysis using simulation techniques appears in the second volume of this report.

Part Two describes recovery algorithms for database systems. A non-mathematical survey of these algorithms is presented in Section VI. It describes methods for: undoing a transaction after it aborts; redoing committed transactions after a site fails; extending any centralized undo or redo algorithm to a distributed system.

A mathematical model of recovery algorithms is presented in Section VII.

Each of the algorithms described in Section VI is proved correct using this model.

Section VIII presents a new algorithm for site recovery in a distributed database system. The problem is how to bring a site's database up to date after it has recovered from failure. The solution allows different portions of the database to be brought up to date independently, thereby avoiding a strongly synchronized bulk transfer of the entire database to the recovered site.

Section IX presents an optimal algorithm for undoing a set of transactions to a consistent state. The problem is that when one transaction is undone, any transaction that read its output must also be undone. Thus, undo's may cascade requiring many other transactions to be undone. It is shown that there is a unique best set of transactions to undo, and a fast algorithm for finding this set is described.

SECTION II

A SOPHISTICATE"S INTRODUCTION TO

DISTRIBUTED DATABASE CONCURRENCY CONTROL*

Philip A. Bernstein

Nathan Goodman

*Published in the Proceedings of the 8th International Conference on Very Large Data Bases, Mexico City, Sept. 1982.

ABSTRACT

Dozens of articles have been published describing "new" concurrency control algorithms for distributed database systems. All of these algorithms can be derived and understood using a few basic concepts. We show how to decompose the concurrency control problem into several subproblems, each of which has just a few known solutions. By appropriately combining known solutions to the subproblems, we show that all published concurrency control algorithms and many new ones can be constructed. The glue that binds the subproblems and solutions together is a mathematical theory known as serializability theory.

This paper does not assume previous knowledge of distributed database concurrency control algorithms, and is suitable for both the uninitiated and the cognoscente.

1. INTRODUCTION

A distributed database system (DDBS) is a database system (DBS) that provides commands to read and write data that is stored at multiple sites of a network. If users access a DDBS concurrently, they may interfere with each other by attempting to read and/or write the same data. Concurrency control is the activity of preventing such behavior.

Dozens of algorithms that solve the DDBS concurrency control problem have been published (see [BG1] and the references). Unfortunately, many of these algorithms are so complex that only an expert can understand them.

To remedy this situation, we have developed a simple framework for understanding concurrency control algorithms. The framework decomposes the problem into subproblems and gives basic techniques for solving each subproblem. To understand a published algorithm, one first identifies the technique used for each subproblem and then checks that the techniques are appropriately combined. The framework can also be used to develop new algorithms by combining existing techniques in new ways.

The paper has 10 sections. Sections 2 and 3 set the stage by describing a simple DDBS architecture and sketching the framework in terms of the architecture. The framework itself appears in Sections 4-8. Section 9 uses the framework to explain several published algorithms. Section 10 is the conclusion.

This paper refines an earlier survey of concurrency control algorithms [BG1]. The earlier paper includes many technical details that are omitted here. We urge the interested reader to consult [BG1] for more details.

2. DISTRIBUTED DBS ARCHITECTURE

We use a simple model of DDBS structure and behavior. The model highlights those aspects of a DDBS that are important for understanding concurrency control, while hiding details that don't affect concurrency control.

A database consists of a set of data items, denoted {...,x,y,z}. In practice, a data item can be file, record, page, etc. But for the purposes of this paper, it's best to think of a data item as a simple variable. For now, assume each data item is stored at exactly one site.

Users access data items by issuing *Read* and *Write* operations. Read(x) returns the current value of x. Write(x,new value) updates the current value of x to new-value.

Users interact with the DDBS by executing programs called transactions. A transaction only interacts with the outside world by issuing Reads and Writes to the DDBS or by doing terminal I/O. We assume that every transaction is a complete and correct computation; each transaction, if executed alone on an initially consistent database, would terminate, produce correct results, and leave the database consistent.

Each site of a DDBS runs one or more of the following software modules (see figures 1 and 2): a transaction manager (TM), a data manager (DM) or a scheduler. Transactions talk to TM's; TM's talk to schedulers; schedulers talk among themselves and also talk to DM's; and DM's manage the data.

Each transaction issues all of its Reads and Writes to a single TM.

A transaction also issues a Begin operation to its TM when it starts executing and an End when it's finished.

The TM forwards each Read and Write to a scheduler. (Which scheduler depends on the concurrency control algorithm; usually, the scheduler is at

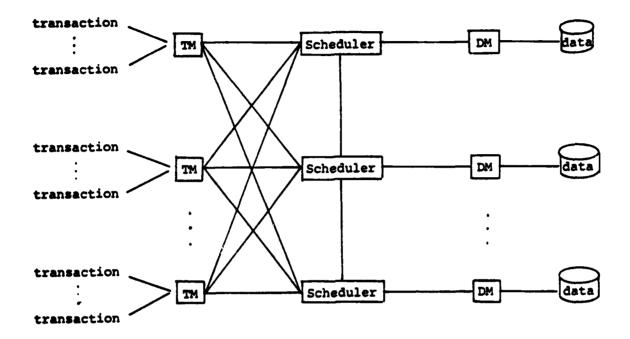


Figure 1
DDBS Architecture

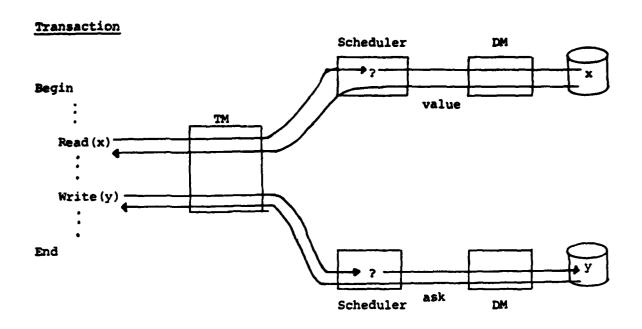


Figure 2
Processing Operations

the same site as the data being read or written. In some algorithms,

Begins and Ends are also sent to schedulers.)

The scheduler controls the order in which DM's process Reads and Writes. When a scheduler receives a Read or Write operation, it can either output the operation right away (usually to a DM, sometimes to another scheduler), delay the operation by holding it for later action, or reject the operation. A rejection causes the system to abort the transaction that issued the operation: every Write processed on behalf of the transaction is undone (restoring the old value of the data item), and every transaction that read a value written by the aborted transaction is also aborted. This phenomenon of one abort triggering other aborts is called cascading aborts. (It is usually avoided in commercial DBS's by not allowing a transaction to read another transaction's output until the DBS is certain that the latter transaction will not abort. In this paper, we will not try to prevent cascading aborts.) This paper does not discuss techniques for implementing abort. See [GMBL, HS, LS].

The DM executes each Read and Write it receives. For Read, the DM looks in its local database and returns the requested value. For Write, the DM modifies its local database and returns an acknowledgment. The DM sends the returned value or acknowledgment to the scheduler, which relays it back to the TM, which relays it back to the transaction.

DM's do not necessarily execute operations first-come-first-served.

If a DM receives a Read(x) and a Write(x) at about the same time, the

DM is free to execute these operations in either order. If the order

matters (as it probably does in this case), it is the scheduler's responsibility to enforce the order. This is done by using a hor sh 'ing communication discipline between schedulers and DM's (see figure 3): If the

To execute Read(x) on behalf of transaction 1
followed by Write(x) on behalf of transaction 2

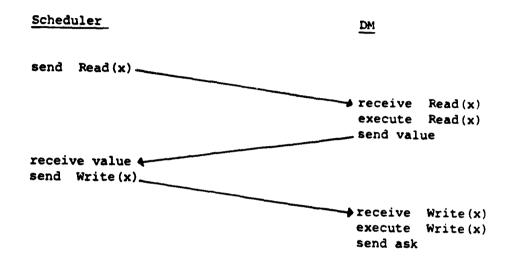


Figure 3
Handshaking

scheduler wants Read(x) to be executed before Write(x), it sends

Read(x) to the DM, waits for the DM's response, and then sends Write(x).

Thus the scheduler doesn't even send Write(x) to the DM until it knows

Read(x) was executed. Of course, when the execution order doesn't matter,

the scheduler can send operations without the handshake.

Handshaking is also used between other modules when execution order is important.

3. THE FRAMEWORK

The DDBS modules most important to concurrency control are schedulers.

A concurrency control algorithm consists of some number of schedulers, running some type of scheduling algorithm, in a centralized or distributed fashion.

In addition, the concurrency control algorithm must handle "replicated data" somehow. (TM's often handle this problem.)

To understand a concurrency control algorithm using our framework one determines

- (i) the type of scheduling algorithm used (discussed in Sections 5 and 8).
- (ii) the location of the scheduler(8), i.e. centralized vs. distributed (Section 6), and
- (iii) how replicated data is handled (Section 7).

The framework also includes rules that tell when a concurrency control algorithm is correct. These rules give precise conditions under which a DDBS produces correct executions. These rules, called **serializability** theory, are discussed in the next section.

4. SERIALIZABILITY THEORY

Serializability theory is a collection of mathematical rules that tell whether a concurrency control algorithm works correctly [BSW, Casa, EGLT, Papa, PBR, SLR]. Serializability theory does its job by looking at the executions allowed by the concurrency control algorithm. The theory gives a precise condition under which an execution is correct. A concurrency control algorithm is then judged to be correct if all of its executions are correct.

4.1 Logs

Serializability theory models executions by a construct called a log.

A log identifies the Read and Write operations executed on behalf of each transaction, and tells the order in which those operations were executed.

Following Lamport, we allow an execution order to be a partial order [Lamp].

A transaction log represents an allowable execution of a single transaction. Formally, a transaction log is a partially ordered set (poset) $T_{i} = (\Sigma_{i}, <_{i}) \quad \text{where} \quad \Sigma_{i} \quad \text{is the set of Reads and Writes issued by (an execution of) transaction i, and <_{i} tells the order in which those operations must be executed. We write transaction logs as diagrams.$

$$T_1 = \frac{r_1(x)}{r_1(z)} w_1(x)$$

 T_1 represents a transaction that reads x and z in parallel, and then writes x. (Presumably, the value written depends on the values read.)

Let $T = \{T_0, ..., T_n\}$ be a set of transaction logs. A DDBS log (or simply a log) over T represents an execution of $T_0, ..., T_n$. Formally, a log over T is a poset $L = (\Sigma, <)$ where

1.
$$\Sigma = U_{i=0}^{n} \Sigma_{i}$$
, and

2.
$$' < \supseteq U_{i=0}^{n} <_{i}$$
.

Conditions (1) states that the DDBS executed all, and only, the operations submitted by T_0, \dots, T_n . Condition (2) states that the DDBS honored all operation orderings stipulated by the transactions.

The following are all possible logs over the example transaction $\log T$, from above.

$$(1) \qquad \qquad \begin{matrix} r_1[x] \\ \downarrow \\ r_1[z] \end{matrix} \qquad \qquad w_1[x]$$

(2)
$$\begin{array}{c} r_1[x] \\ \downarrow \\ r_1[z] \end{array}$$
 $w_1[x]$

$$\begin{array}{c}
\mathbf{r_1}[\mathbf{x}] \\
\mathbf{r_1}[\mathbf{z}]
\end{array}$$

Notice that the DDBS is not required to process $\operatorname{Read}(x)$ and $\operatorname{Read}(z)$ in parallel, even though T_1 allows this parallellism. However, the DDBS is not allowed to reverse or eliminate any ordering stipulated by T_1 . The following is not a log over T_1 .

(4)
$$r_1[x] \sim w_1[x]$$
 $r_1[z]$

because it reverses the order in which T_1 reads and writes x.

There is one further constraint on the form of logs. Two operations conflict if they operate on the same data item and (at least) one of them is a Write. To ensure that logs represent unique computations, we require that all pairs of conflicting operations be ordered. This constraint applies to transaction logs as well as DDBS logs.

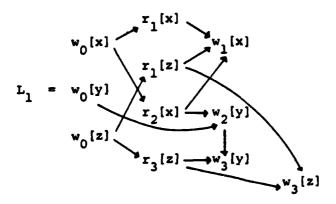
We use $r_i[x]$ (resp., $w_i[x]$) to denote a Read (resp., Write) on x issued by T_i . To keep this notation unambiguous, we assume that no transaction reads or writes a data item more than once.

Given transaction logs

$$T_0 = w_0[y]$$
 $w_0[z]$
 $T_1 = x_1[x]$
 $x_1[x]$

$$T_2 = r_2[x] + w_2[y]$$
 $T_3 = r_3[z] < w_3[y]$
 $w_3[y]$

the following is a log over $\{T_0, T_1, T_2, T_3\}$.



(Note that orderings implied by transitivity are usually not drawn. E.g. $w_0[y] < w_3[y]$ is not drawn in the diagram, although it follows from $w_0[y] < w_2[y]$ and $w_2[y] < w_3[y]$.)

4.2 Log Equivalence

Let L be a log over some set T, and suppose $w_i[x]$ and $r_j[x]$ are operations in L. We say $r_j[x]$ reads-from $w_i[x]$ if $w_i[x] < r_j[x]$ and no $w_k[x]$ falls between $r_i[x]$ and $w_j[x]$. In this log

$$w_0[x] + r_1[x] + w_2[x] + r_3[x] + r_4[x]$$

 $r_1[x]$ reads-from $w_0[x]$, and $r_3[x]$ and $r_4[x]$ read-from $w_2[x]$. We call $w_i[x]$ a final-write in L if no $w_k[x]$ follows it. In this log

$$w_0[x] + w_1[x] + w_2[y] + r_2[y]$$

 $w_1[x]$ and $w_2[y]$ are final-writes.

Intuitively, two logs over T are equivalent if they represent the same computation. Formally, two logs over T are equivalent if

- (1) each Read reads-from the same Write in both logs, and
- (2) they have the same final-writes.

Condition (1) ensures that each transaction reads the same values from the database in each log. Condition (2) ensures that the same transaction writes the final value of a given data item in both logs.

The following $\log L_2$ is equivalent to $\log L_1$ of Section 4.2.

(When we write a log as a sequence, e.g. L_2 , we mean that the log is totally ordered: each operation precedes the next one and all subsequent ones in the sequence. Thus, in L_2 , $w_0[x] < w_0[y] < w_0[z] < r_2[x]$ )

4.3 Serializable Logs

A serial log is a total order on Σ such that for every pair of transactions T_i and T_j , either all of T_i 's operations precede all of T_j 's, or vice versa (e.g., L_2 in Section 4.2). A serial log represents an execution in which there is no concurrency whatsoever; each transaction executes from beginning to end before the next transaction begins. From the point of view of concurrency control, therefore, every serial log represents an obviously correct execution.

What other logs represent correct executions? From the point of view of concurrency control, a correct execution is one in which concurrency is invisible. That is, an execution is correct if it is equivalent to an execution in which there is no concurrency. Serial logs represent the latter executions, and so a correct log is any log equivalent to a serial log. Such logs are termed serializable (SR). Log L_1 of Sec. 4.1 is SR, because it is equivalent to serial log L_2 of Sec. 4.2. Therefore L_1 is a correct log.

Serializability theory is the study of serializable logs.

4.4 The Serializability Theorem

This section presents the main theorem of serializability theory. Later sections rely on this theorem to analyze concurrency control algorithms.

This theorem uses a graph derived from a log, called a serialization graph.

Suppose L is a log over $\{T_0, \ldots, T_n\}$. The serialization graph for L, SG(L), is a directed graph whose nodes are T_0, \ldots, T_n , and whose edges are all $T_i + T_j$ such that, for some x, either (i) $r_i[x] < w_j[x]$, or (iii) $w_i[x] < r_i[x]$, or (iiii) $w_i[x] < w_i[x]$. The serialization graphs for

example log L, is

$$SG(L_1) = T_0 \xrightarrow{T_1} T_3$$

Edge $T_0 + T_1$ is present because $w_0[x] < r_1[x]$. Edge $T_2 + T_1$ is caused by $r_2[x] < w_1[x]$. Edge $T_2 + T_3$ arises from $w_2[y] < w_3[y]$. And so forth.

SERIALIZABILITY THEOREM. If SG(L) is acyclic then L is SR.

For example, since $SG(L_1)$ is acyclic, L_1 is SR.

We can also use the Serializability Theorem to determine if a scheduler produces SR logs. First, we characterize the logs produced by the scheduler. Then we prove that every such log has an acyclic SG [BSW, Papa].

Some concurrency control algorithms schedule read-write conflicts separately from write-write conflicts. It is easier to analyze such algorithms using a restatement of the Serializability Theorem. Define the read-write serialization graph for L, $SG_{TW}(L)$, as follows: $SG_{TW}(L)$ has nodes T_0, \ldots, T_n and edges $T_i + T_j$ such that, for some x, either (i) $r_i[x] < w_j[x]$, or (ii) $w_i[x] < r_j[x]$. In other words, $SG_{TW}(L)$ is like SG(L) except we don't care about write-write conflicts. The write-write serialization graph for L, $SG_{WW}(L)$, is defined analogously: the nodes are T_0, \ldots, T_n , and the edges are $T_i + T_j$ such that, for some x, $w_i[x] < w_j[x]$.

$$SG_{rw}(L_1) = T_0 \xrightarrow{T_1} T_3 \qquad SG_{ww}(L_1) = T_0 \xrightarrow{T_1} T_3$$

Of course, $SG(L) = SG_{rw}(L) \cup SG_{rw}(L)$.

RESTATED SERIALIZABILITY THEOREM [BG1]. If the following four conditions hold, then L is SR

- (i) SG__ (L) is acyclic.
- (ii) SG (L) is acyclic.
- (iii) For all T_i and T_j , if T_i precedes T_j in $SG_{TW}(L)$ then either T_i precedes T_j in $SG_{WW}(L)$, or there is no path between them in $SG_{TW}(L)$.
- (iv) For all T_i and T_j , if T_i precedes T_j in $SG_{ww}(L)$ then either T_i precedes T_j in $SG_{rw}(L)$, or there is no path between them in $SG_{rw}(L)$.

Conditions (i)-(iv) are just another way of saying that SG(L) is acyclic. The conditions allow us to analyze the correctness of read-write (rw) scheduling almost independently of write-wirte (ww) scheduling.

5. SCHEDULERS

There are four types of schedulers for producing SR executions: twophase locking, timestamp ordering, serialization graph checking and
certifiers. Each type of scheduler can be used to schedule rw conflicts,
ww conflicts, or both. This section describes each type of scheduler
assuming it is used for both kinds of conflict. Ways of combining scheduler
types (e.g. two-phase locking for rw conflicts and timestamp ordering
for ww conflicts) are described in Section 9. This section also assumes
that the scheduler runs at a single site, see figure 4; Section 5 lifts
this restriction.

5.1 Two-Phase Locking

- A two-phase locking (2PL) scheduler is defined by three rules [EGLT]:
 - i. Before outputting $r_i[x]$ (resp. $w_i[x]$), set a read-lock (resp. write-lock) for T_i on x. The lock must be held (at least) until the operation is executed by the appropriate DM. (Handshaking can be used to guarantee that locks are held long enough.)
 - ii. Different transactions cannot simultaneously hold "conflicting"
 locks. Two locks conflict if they are on the same data item and
 (at least) one is a write-lock. If rw and ww scheduling is
 done separately, the definition of "conflict" is modified. For
 rw scheduling, two locks on the same data item conflict if
 exactly one is a write-lock; i.e., write-locks don't conflict
 with each other. For ww scheduling, both locks must be write-locks.
- iii. After releasing a lock, a transaction cannot obtain any more locks.

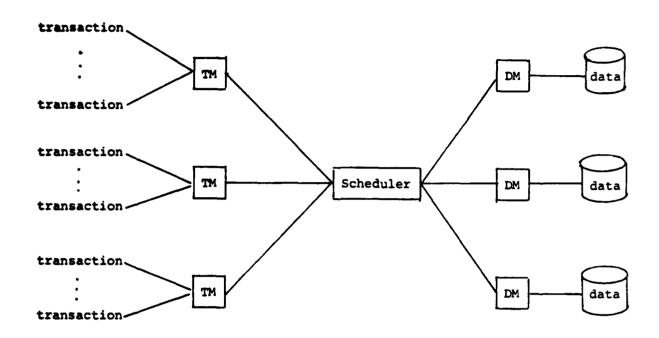


Figure 4

DDBS Architecture With Centralized Scheduler

Rule (iii) causes locks to be obtained in a two-phase manner. During its growing phase, a transaction obtains locks without releasing any. By releasing a lock, the transaction enters its shrinking phase during which it can only release locks. Rule (iii) is usually implemented by holding all of a transaction's locks until it terminates.

2PL THEOREM. A 2PL scheduler only produces SR logs.

Proof Sketch. Consider a log L produced by a 2PL scheduler. If $T_i^+T_j^-$ is in SG(L), then T_i^- released some lock before T_j^- obtained that lock. If there's a nonempty path in SG(L) from T_i^- to T_i^- (i.e., a cycle) then, by transitivity, T_i^- released a lock before T_i^- obtained some lock, thereby breaking rule (iii). So, SG(L) is acyclic. By the Serializability Theorem, this implies that L is SR.

Due to rule (ii), an operation received by a scheduler may be delayed because another transaction already owns a conflicting lock. Such blocking situations can lead to deadlock. For example, suppose $r_1[x]$ and $r_2[y]$ set read-locks, and then the scheduler receives $w_1[y]$ and $w_2[x]$. The scheduler cannot set the write-lock needed by $w_1[y]$ because T_2 holds a read-lock on y. Nor can it set the write-lock for $w_2[x]$ because T_1 holds a read-lock on x. And, neither T_1 nor T_2 can release its read-lock before getting the needed write-lock because of rule (iii). Hence, we have a deadlock: T_1 is waiting for T_2 which is waiting for T_1 .

Deadlocks can be characterized by a waits-for graph [Holt, KC], a directed graph whose nodes represent transactions and whose edges represent waiting relationships: edge $T_i + T_j$ means T_i is waiting for a lock owned by T_j . A deadlock exists if and only if (iff) the waits-for graph has a cycle. E.g., in the above example the waits-for graph is

$$T_1$$
 .

A popular way of handling deadlock is to maintain the waits-for graph and periodically search it for cycles. (See [Chap. 5, AHU] for cycle detection algorithms.) When a deadlock is detected, one of the transactions on the cycle is aborted and restarted, thereby breaking the deadlock.

5.2 Timestamp Ordering

In timestamp ordering (T/O) each transaction is assigned a globally unique timestamp by its TM. (See [BG1, Thom] for how this is done.) The TM attaches the timestamp to all operations issued by the transaction. A T/O scheduler is defined by a single rule: Output all pairs of conflicting operations in timestamp order. Make sure conflicting operations are executed by DMs in the order they were output. (Handshaking can be used to make sure of this.) As for 2PL, the definition of "conflicting operation" is modified, if rw and www scheduling are done separately.

T/O THEOREM. A T/O scheduler only produces SR logs.

Proof Sketch. Since every pair of conflicting operations is in time-stamp order, each edge $T_i + T_j$ in SG has $TS(T_i) < TS(T_j)$ (where $TS(T_i)$ is the timestamp of T_i). Thus, SG cannot have any cycles. So, by the Serializability Theorem, the log produced by the scheduler is SR.

Several varieties of T/O schedulers have been proposed. We only sketch these variations here. Full details appear in [BG1].

A basic T/O scheduler outputs operations in essentially first-comefirst-served order, as long as the T/O scheduling rule holds. When the scheduler receives $r_*[x]$ it does the following.

if TS(i) < largest timestamp of any Write on x yet "accepted" $then reject <math>r_i[x]$

else "accept" $r_i[X]$ and output it as soon as all Writes on x with smaller timestamp have been acknowledged by the DM.

When the scheduler receives $w_{i}[x]$ it behaves as follows.

if TS(i) < largest timestamp of any Read or Write on x yet "accepted"

then reject w, [x]

else "accept" $w_i[x]$ and output it as soon as all Reads and Writes on x with smaller timestamp have been acknowledged by the DM.

A conservative T/O scheduler avoids rejections by delaying operations instead. An operation is delayed until the scheduler is sure that outputting it will cause no future operations to be rejected. Conservative T/O requires that each scheduler receive Reads and Writes from each TM in timestamp order. To output any operation, the scheduler must have an operation from each TM in its "input queue." The scheduler then "accepts" the operation with smallest timestamp. "Accept" means remove the operation from the input queue, and output it as soon as all conflicting operations with smaller timestamp have been acknowledged by the DM. Variations on conservative T/O are discussed in [BG1, BSR].

Basic T/O and conservative T/O are endpoints of a spectrum. Basic T/O delays operations very little, but tends to reject many operations. Conservative T/O never rejects operations, but tends to delay them a lot. One can imagine T/O schedulers between these extremes. To our knowledge, no one has yet proposed such a scheduler.

Thomas'write rule (TWR) is a technique that reduces delay and rejection [Thom]. TWR can only be used to schedule Writes, and needs to be combined with basic or conservative T/O to yield a complete scheduler.

If we're only interested in www scheduling, TWR is simple. When the scheduler receives w. [x] it does the following.

if TS(i) < largest timestamp of any Write or x yet "accepted"
then "pretend" to execute w_i[x]--i.e., send an acknowledgement
back to the TM, but don't send the Write to the DM
else "accept" w_i[x] and process it as usual.

The basic T/O-TWR combination works like this. Reads are processed exactly as in basic T/O. But when the scheduler receives a $w_i[x]$, it combines the basic T/O rule and TWR as follows.

then "pretend" to execute W_i[x]

5.3 Serialization Graph Checking

This type of scheduler works by explicitly building a serialization graph, SG, and checking it for cycles. Like basic T/O, an SG checking scheduler never delays an operation (except for handshaking reasons).

Rejection is the only action used to avoid incorrect logs.

An SG checking scheduler is defined by the following rules.

- When transaction T_i Begins, add node T_i to SG.
- ii. When a Read or Write from T_i is received, add all edges $T_j + T_i$ such that T_j is a node of SG, and the scheduler has already output a conflicting operation from T_j . As for the previous schedulers, the definition of "conflicting operation" is modified if T_j and T_j are scheduled separately.

iii. If after step (ii) SG is still acyclic, output the operation.

Make sure that conflicting operations are executed by DM's in the order they were output. (Handshaking can be used for this.)

iv. If after (ii) SG has become cyclic, reject the operation. Delete node T_i and all edges $T_i + T_j$ or $T_j + T_i$ from SG. (SG is now acyclic again.)

SG CHECKER THEOREM. An SG checking scheduler only produces SR logs.

Proof sketch. Every log produced by the scheduler has an acyclic SG.
So, by the Serializibility Theorem, every log is SR.

One technical problem with SG checkers is that a transaction must remain in SG even after it has terminated. A transaction can only be deleted from SG when it is a source node of the graph, i.e., when it has no incoming edges. See [Casa] for a discussion of this problem and techniques for efficiently encoding information about terminated transactions that remain in SG.

5.4 Certifiers

The term "certifier" refers to a scheduling philosophy, not a specific scheduling rule. A certifier is a scheduler that makes its decisions on a per-transaction basis. When a certifier receives an operation, it internally stores information about the operation and outputs it as soon as all earlier conflicting operations have been acknowledged. When a transaction ends, its TM sends the End operation to the certifier. At this point, the certifier checks its stored information to see if the transaction executed serializably. If it did, the certifier certifies the transaction, allowing it to terminate; otherwise, the certifier aborts the transaction.

All of the earlier schedulers can be adapted to work as certifiers. Here is an SG checking certifier. When the certifier receives an operation, it adds a node and some edges to SG as explained in the previous section. The certifier does not check for cycles at this time. When a transaction, T_i , ends, the certifier checks SG for cycles. If T_i does not lie on a cycle, it is certified; otherwise it is aborted.

SG CERTIFIER THEOREM. An SG checking certifier only produces SR logs.

Proof sketch. Consider any "completed" log produced by the certifier.

Completed means that all uncertified transactions are aborted and removed from the log. (As always, any transaction that read data written by an aborted transaction is also aborted; this may include some certified transaction.) The completed log has an acyclic serialization graph. So by the Serializability Theorem, the log is SR.

Here is a 2PL certifier [Thom, KR]. Define a transaction to be active from the time the certifier receives its first operation until the certifier

processes its End. The certifier stores two sets for each active transaction \mathbf{T}_i :

 T_i 's readset, $RS(i) = \{x | \text{the certifier has output } r_i[x] \}$

 T_i 's writeset, $WS(i) = \{x | \text{the certifier has output } w_i[x] \}$.

The certifier updates these sets as it receives operations. When the certifier receives End,, it runs the following test.

Let $RS(active) = U(RS(j), such that T_j is active, but j \neq i)$ $WS(active) = U(WS(j), such that T_j is active, but j \neq i)$

if RS(i) \cap WS(active) $\neq \emptyset$, or

WS(i) \cap (RS(active) UWS(active)) $\neq \emptyset$

then certify Ti

else abort T.

This amounts to pretending that transactions hold imaginary locks on their readsets and writesets. When transaction T_i ends, the certifier sees if T_i 's imaginary locks conflict with the imaginary locks held by other active transactions. If there is no conflict, T_i is certified; else T_i is aborted.

2PL CERTIFIER THEOREM. A 2PL certifier only produces SR logs.

Proof sketch. Consider a completed log L produced by the certifier.

If $T_i + T_j$ is in SG(L), then since both T_i and T_j were certified, the certifier processed End, before End,. If there's a nonempty path in SG(L) from T_i to T_i (i.e., a cycle) then, by transitivity, the certifier processed End, before End. This is absurd. So, SG(L) is acyclic, and by the Serializability Theorem, L is SR.

T/O certifiers are also possible. To our knowledge, no one has proposed this algorithm yet.

Certifiers can also be built that check for serializable executions during transactions' executions, not just at the end. The extra version of this idea is to check for serializability on every operation. At this extreme, the certifier reduces to a "normal" scheduler.

6. SCHEDULER LOCATION

The schedulers of Sect: on 5 can be modified to work in a distributed manner. Instead of one scheduler for the whole system, we now assume one scheduler per DM (refer back to figure 1). The scheduler normally runs at the same site as the DM, and schedules all operations that the DM executes.

The new issue in this setting is that the distributed schedulers must cooperate to attain the scheduling rules of Section 5.

The main problem caused by distributing schedulers is the maintenance of global data structures. Distributed 2PL schedulers need a global waits-for graph. Distributed SG checkers need a global SG. In distributed T/O scheduling, no global data structures are needed; each scheduler can make its scheduling decisions using local copies of R-TS(x) and W-TS(x) for each x at its DM. Distributed certifiers generally manifest the same problems as their corresponding schedulers.

6.1 Distributed Two Phase Locking

Refer to the 2 PL scheduling rules of Section 5.1. Rules (i) and (ii) are "local." The scheduler for data item x schedules all operations on x. Hence this scheduler can set all locks on x. Rule (iii) requires a small amount of inter-scheduler cooperation: no scheduler can obtain a lock for transaction T_i after any scheduler releases a lock for T_i . This can be done by handshaking between TMs and schedulers. When T_i Ends, its TM waits until all of T_i 's Reads and Writes are acknowledged. At this point the TM knows that all of T_i 's locks are set, and it's safe to release locks. The TM forwards End_i to the schedulers which then release T_i 's locks.

One problem with distributed 2PL is that multi-site deadlocks are possible. Suppose x and y are stored at sites A and B, respectively. Suppose $r_i[x]$ is processed at A, setting a read-lock on x for T_i at A; and $r_j[y]$ is processed at B, setting a read-lock on y for T_j at B. If $w_j[x]$ and $w_i[y]$ are now issued, a deadlock will result; T_j will be waiting for T_i to release its lock on x at A and T_i will be waiting for T_j to release its lock on y at B. Unfortunately, the deadlock isn't apparent by looking at site A or B alone. Only by taking the union of the waits-for graphs at both sites does the deadlock cycle materialize.

See [MM, Ston, GlSh, RSL] for solutions to this problem.

6.2 Distributed Timestamp Ordering

T/O schedulers are easy to distribute, because the T/O scheduling rule of Section 5.2 is inherently local. Consider a basic T/O scheduler for data item x. To process an operation on x, the scheduler only needs to know if a conflicting operation with larger timestamp has been accepted. Since the scheduler handles all operations on x, it can make this decision itself.

6.3 Distributed Serialization Graph Checking

SG checkers are harder to distribute than the other scheduler because the semialization graph, SG, is inherently global. A transaction that accesses data at a single site can become involved in a cycle spanning many sites. See [Casa] for a discussion of this problem.

6.4 Distributed Certifiers

Distributed certifiers have a synchronization requirement a little like rule (iii) of 2PL: T_i 's TM must not send End_i to any certifier, until all of T_i 's Reads and Writes have been acknowledged. I.e., we must not try to certify T_i at any site until we are ready to certify T_i at all sites.

Beyond this, each distributed certifier behaves like the corresponding scheduler. A distributed 2PL certifier needs little inter-scheduler cooperation (beyond the previous paragraph). The certifier at each site keeps track of the data items at its site read or written by active transactions. When the certifier at site A receives End_i, it sees if any active transaction conflicts with T_i at site A. If not, T_i is certified at site A. If not, T_i is certified at then it is "really" certified; else T_i is aborted.

A distributed SG certifier shares the problems of distributed SG schedulers: the certifier needs to check for cycles in a global graph every time a transaction ends.

6.5 Other Architectures

Centralized and distributed scheduling are endpoints of a spectrum. One can imagine hybrid architectures with multiple DM's per scheduler. See figure 5. This architecture adds no technical issues beyond those already discussed.

Hierarchical scheduler architectures are also possible. See figure 6.

To our knowledge, no one has studied this approach yet.

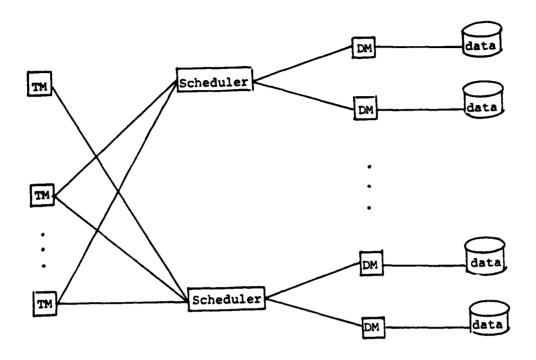


Figure 5
Hybrid Architecture

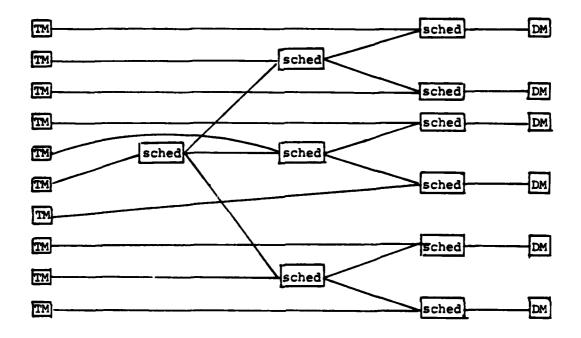
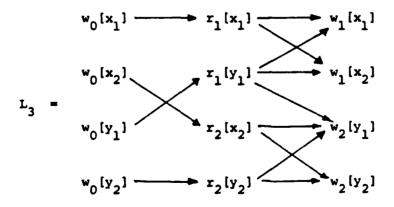


Figure 6
Hierarchical Architecture

DATA REPLICATION

In a replicated database, each logical data item, x, can have many 'ysical copies, denoted $\{x_1,\ldots,x_m\}$, which are resident at different M's. Transactions issue Reads and Writes on logical data items. TM's ranslate those operations into Reads and Writes on physical data. The ffect, as seen by each transaction, must be as if there were only one copy of each data item.

There is a simple way to obtain this effect. Each TM translates $r_i[x]$ into $r_i[x_j]$ for some copy x_j of x and $w_i[x]$ into $\{w_i[x_j] | all copies <math>x_i$ of x. If the scheduler(s) is SR, the effect is just like a nonreplicated database. To see this, consider a serial log equivalent to the SR log that executed. Since each transaction writes into all copies of each logical data item, each $r_i[x_j]$ reads from the 'latest' transaction preceding it that wrote into any copy of x. But this is exactly what would have happened had there been only one copy of x. (For a more rigorous explanation, see [ABG].) Consider this example.



 \mathbf{x}_1 and \mathbf{x}_2 are copies of logical data item \mathbf{x} ; \mathbf{y}_1 and \mathbf{y}_2 are copies of \mathbf{y} . \mathbf{T}_0 produces initial values for both copies of each data item. \mathbf{T}_1 reads \mathbf{x} and \mathbf{y} , and writes \mathbf{x} ; \mathbf{T}_2 reads \mathbf{x} and \mathbf{y} , and writes \mathbf{y} .

 L_{χ} is SR. It is equivalent to the following serial log:

$$\mathbf{L}_{4} = \mathbf{w}_{0}[\mathbf{x}_{1}]\mathbf{w}_{0}[\mathbf{x}_{2}]\mathbf{w}_{0}[\mathbf{y}_{1}]\mathbf{w}_{0}[\mathbf{y}_{2}]\mathbf{r}_{1}[\mathbf{x}_{1}]\mathbf{r}_{1}[\mathbf{y}_{1}]\mathbf{w}_{1}[\mathbf{x}_{1}]\mathbf{w}_{1}[\mathbf{x}_{2}]\mathbf{r}_{2}[\mathbf{x}_{2}]\mathbf{r}_{2}[\mathbf{y}_{2}]\mathbf{w}_{2}[\mathbf{y}_{1}]\mathbf{w}_{2}[\mathbf{y}_{2}] .$$

Note that each Read, e.g. $r_2[x_2]$ or $r_2[y_2]$, reads from the 'latest' transaction preceding it that wrote into any copy of the data item. Therefore, L_4 has the same effect as the following log in which there is no replicated data:

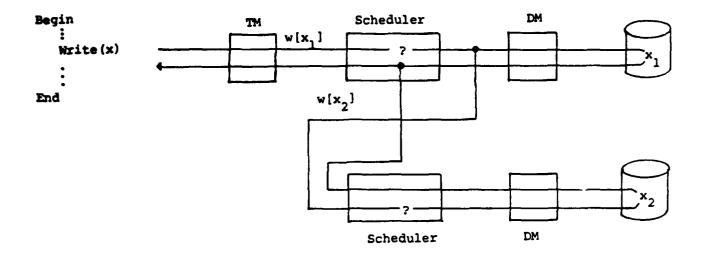
$$L_4^t = w_0[x]w_0[y]r_1[x]r_1[y]w_1[x]r_2[x]r_2[y]w_2[y]$$
.

We call this the *do nothing* approach to replication-just write into all copies of each data item and use an SR scheduler.

Two other approaches to replication have been suggested. In the primary copy approach, and copy of each x, say x_p , is designated its primary copy [Ston]. Each TM translates $r_i[x]$ into $r_i[x_j]$ for some copy x_j , as before. Writes are translated differently, though. The TM translates $w_i[x]$ into a single Write, $w_i[x_p]$, on the primary copy. When the primary copy's scheduler outputs $w_i[x_p]$, it also issues Writes on the other copies of x (i.e., $w_i[x_1], \ldots, w_i[x_m]$). See Figure 7. These Writes are processed by the schedulers for x_1, \ldots, x_m in the usual way. For example, in 2PL, the scheduler for x_j must get a write-lock on x_j for T_i before outputting $w_i[x_j]$. The primary copy's scheduler may be centralized (in which case the technique is called primary site [AD]), or distributed with the primary copy's DM.

Primary copy is a good idea for 2PL schedulers. It eliminates the possibility of deadlock caused by Writes on different copies of one data

Transaction



Note: x₁ is primary copy.

Figure 7
Processing Writes In Primary Copy

item. Suppose x has copies x_1 and x_2 . Suppose T_1 and T_2 want to Write x at about the same time. In the do nothing approach, the following execution is possible: T_1 locks x_1 ; T_2 locks x_2 ; T_1 tries to lock x_2 but is blocked by T_2 's lock; T_2 tries to lock x_1 but is blocked by T_1 's lock. This is a deadlock. Primary copy avoids this possibility because each transaction must lock the primary copy first.

In the *voting* approach to replication, TM's again distribute Writes to all copies of each data item [Thom]. Assume we are using distributed schedulers. When a scheduler is ready to output $w_i[x_j]$, it sends a vote of *yes* to the *vote collector* for x; it does not output $w_i[x_j]$ at this time. When the vote collector receives yes votes from a majority of schedulers, it tells *all* schedulers to output their Writes. (Each scheduler may need to update its local data structures before outputting $w_i[x_j]$, e.g. set a write-lock on x_i .) Assume each scheduler is correct (i.e., produces an acyclic SG). Then, since every pair of conflicting operations was voted yes by some correct scheduler (both operations got a majority of yes's), the SG must be acyclic and the result is correct.

The principal benefit of voting is fault tolerance; it works correctly as long as a majority of sites holding a copy of x are running. See [Thom, Giff] for details.

8. MULTIVERSION DATA

Let us return to a database system model where each logical data item is stored at one DM.

In a multiversion database each Write, $w_i[x]$, produces a new copy (or version) of x, denoted x^i . Thus, the value of x is a set of versions. For each Read, $r_i[x]$, the scheduler selects one of the versions of x to be read. Since Writes don't overwrite each other, and since Reads can read any version, the scheduler has more flexibility in controlling the effective order of Reads and Writes.

Although the database has multiple versions, users expect their transactions to behave as if there were just one copy of each data item. Serial logs don't always behave this way. For example,

$$w_0[x^0]x_1[x^0]w_1[x^1y^1]x_2[x^0y^1]w_2[y^2]$$

is a serial log, but its behavior cannot be reproduced with only one copy of x. We must therefore restrict the set of allowable serial logs.

A serial log is 1-copy serial (or 1-serial) if each $r_1[x]$ reads from the last transaction preceding it that wrote into any version of x. The above log is not 1-serial, because r_2 reads x from w_0 , but $w_0[x^0] < w_1[x^1] < r_2[x^0]$. A log is 1-serializable (1-SR) if it's equivalent to a 1-serial log. 1-serializability is our correctness criterion for multiversion database systems.

All multiversion concurrency control algorithms (that we know of) totally order the versions of each data item in some simple way. A version order, <<, for L is an order relation over versions such that, for each x, << totally orders the versions of x.

Given a version order <<, we define the multiversion SG w.r.t. L and << (denoted MVSG(L,<<)) as SG(L) with the following edges added: for each $r[x^j]$ and $w_k[x^k]$ in L, if $x^k << x^j$ then include $T_k + T_j$, else include $T_i + T_k$.

MULTIVERSION THEOREM [BG3]. A multiversion log is 1-SR iff there

exists a version order << such that MVSG(L,<<) is acyclic.

This theorem enables us to prove multiversion concurrency control algorithms to be correct. We must argue that for every log L produced by the algorithm, MVSG(L, <<) is acyclic for some <<.

The types of multiversion schedulers that have been proposed fall into two classes that approximately correspond to timestamping and locking.

8.1 Multiversion Timestamping

Multiversion concurrency control was first introduced by Reed in his multiversion timestamping method [Reed]. In Reed's algorithm, each transaction has a unique timestamp. Each Read and Write carries the timestamp of the transaction that issued it, and each version carries the timestamp of the transaction that wrote it. The version order is defined by $x^{i} << x^{j}$ if TS(i) < TS(j).

Operations are processed first-come-first served. However, the version selection rules ensure that the overall effect is as if operations were

Note that two operations conflict (and produce an edge in SG(L)) if they operate on the same version and one of them is a write.

Handshaking is used to ensure that logically conflicting operations are executed by DM's in the order the scheduler output them.

processed in timestamp order. To process $r_i[x]$, the scheduler (or DM) returns the version of x with largest timestamp $\leq TS(i)$. To process $w_i[x]$, version x^i is created, unless some $r_j[x]$ has already been processed with $TS(j) \leq TS(i) \leq TS(k)$. If this condition holds, the Write is rejected.

An analysis of MVSG(L,<<) for any L produced by this method shows that every edge $T_i + T_j$ is in timestamp order (TS(i) < TS(j)). Thus MVSG(L,<<) is acyclic, and so L is 1-SR.

8.2 Multiversion Locking

In multiversion locking, the Writes on each data item, x, must be ordered. We define $x^i << x^j$ if $w_i[x^i] < w_j[x^j]$. Each version is in the certified or uncertified state. When a version is first written, it is uncertified. Each Read, $r_i[x]$, reads either the last (wrt <<) certified version of x or any uncertified version of x. When a transaction finishes executing, the database system attempts to certify it. To certify T_i , three conditions must hold:

- C1. For each $r_i[x^j]$, x^j is certified.
- C2. For each $w_i[x^i]$, all $x^j \ll x^i$ are certified.
- C3. For each $w_i[x^i]$ and each $x^j << x^i$, all transactions that read x^j have been certified.

These conditions must be tested atomically. When they hold, T is declared to be certified and all versions it wrote are (atomically) certified.

All analysis of MVSG(L,<<) for any L produced by this method shows that every edge $T_i \to T_j$ is consistent with the order in which transactions were certified. Since certification is an atomic event, the certification

order is a total order. Thus, MVSG(L,<<) is acyclic, and so L is 1-SR.

Two details of the algorithm require some discussion. First, the algorithm can deadlock. For example, in this log

$$w_0[x^0]r_1[x^0]r_2[x^0]w_1[x^1]w_2[x^2]$$

 T_1 and T_2 are deadlocked due to certification condition C3. As in 2PL, deadlocks can be detected by cycle detection on a waits-for graph whose edges include $T_i \to T_j$ such that T_i is waiting for T_j to become certified (so that T_i will satisfy C1-C3).

Second, C1-C3 can be tested atomically without using a critical section. Once C1 or C2 is satisfied for some $r_i[x^j]$ or $w_i[x^i]$, no future event can falsify it. When C3 becomes true for some $w_i[x^i]$, we "lock" x^i so that no future reads can read versions that precede x^i . This allows C1-C3 to be checked one data item at a time. Of course, the waits-for graph must be extended to account for these new version locks.

Two similar multiversion locking algorithms have been proposed which allow at most one uncertified version of each data item. In Stearns' and Rosenkrantz's method [SR], the waits-for graph is avoided by using a timestamp-based deadlock avoidance scheme. In Bayer et al's method [BHR, BEHR], a waits-for graph is used to help prevent deadlocks. This algorithm consults the waits-for graph before selecting a version to read, and always selects a version that creates no cycles.

Multiversion locking algorithms in which queries (read-only transactions) are given special treatment are described in [Dubo], [BG4].

9. COMBINING THE TECHNIQUES

The techniques described in Sections 4-8 can be combined in almost all possible ways. The three basic scheduling techniques (2PL, T/O, SG checking) can be used in scheduler mode or certifier mode. This gives six basic concurrency control techniques. Each technique can be used for rw or ww scheduling or both $(6^2 = 36)$. Schedulers can be centralized or distributed $(36 \times 2 = 72)$, and replicated data can be handled in three ways (Do Nothing, Primary Copy, Voting) $(72 \times 3 = 216)$. Then, one can use multiversions or not $(216 \times 2 = 432)$. By considering the multivarious variations of each technique, the number of distinct algorithms is in the thousands.

To illustrate our framework, we describe some of these algorithms that have already appeared in the literature.

The distributed locking algorithm proposed for System R* uses a 2PL scheduler for rw and ww synchronization. The schedulers are distributed at the DM's. Replication is handled by the do nothing approach.

Distributed INGRES uses a similar locking algorithm [Ston]. The main difference is that distributed INGRES uses primary copy for replication.

Many researchers have proposed algorithms that use conservative T/O for all scheduling [SM, Lela, KNTH, CB]. They typically distribute the schedulers at DM's and take the do nothing approach to replication.

SDD-1 uses conservative T/O for rw scheduling and Thomas' write rule for ww scheduling. The algorithm has distributed schedulers and takes the do nothing approach to replication [BSR]. SDD-1 also uses conflict graph analysis, a technique for preanalyzing transactions to determine which run-time conflicts need not be synchronized.

A method using 2PL for rw scheduling and Thomas' write rule for www scheduling is described in [BGL]. Distributed schedulers and the do nothing approach to replication were suggested. To ensure that the locking order is consistent with the timestamp order, one can use a Lamport clock: Each message is timestamped with the local clock time when it was sent; if a site receives a message with a timestamp, TS, greater than its local clock time, the site pushes its clock ahead to TS. After a transaction obtains all of its locks, it is assigned a timestamp using the TM's local Lamport clock.

Thomas'majority consensus algorithm was one of the first distributed concurrency control algorithms. It uses a 2PL certifier for rw scheduling and Thomas'write rule for ww scheduling. Schedulers are distributed and voting is used for replication. Each transaction is assigned a timestamp from a Lamport clock when it is certified. This ensures that the certification order (produced by rw scheduling) is consistent with the timestamp order used for ww scheduling.

Each of these algorithms is quite complex. A complete treatment of each would be lengthy. Yet by understanding the basic techniques and how they can be correctly combined, we can explain the essentials of each algorithm in a few sentences.

10. PERFORMANCE

Given that thousands of concurrency control algorithms are conceivable, which one is best for each type of application? Every concurrency control algorithm delays and/or aborts some transactions, when conflicting operations are submitted concurrently. The question is: which algorithms increase overall transaction response time the least?

Although there have been several performance studies of some of these algorithms, the results are still inconclusive [GS, GM1, GM2, Lin, LN]. There is some evidence that 2PL schedulers perform well at low to moderate intensity of conflicting operations. However, we know of no quantitative results that tell when 2PL thrashes due to too many deadlocks. There are similar gaps in our understanding of the performance of other types of schedulers. More analysis is badly needed to help us learn how to predict which concurrency control algorithms will perform well for the applications and systems we will encounter in practice.

REFERENCES

- [AD] Alsberg, P.A. and Day, J.D. "A Principle for Resilient Sharing of Distributed Resources," Proc. 2nd Int. Conference on Software Engineering, October 1976.
- [AHU] Aho, A.V., Hopcroft, E., Ullman, J.D. The Design and Analysis of Computer Algorithms, Addison-Wesley Publishing Co. (1975).
- [ABG] Attar, R., P.A. Bernstein, and N. Goodman. "Site Initialization, Recovery, and Backup in a Distributed Database System," Proc. 1982 Berkeley Workshop on Distributed Databases and Computer Networks.
- [Badal] Badal, D.Z. "Correctness of Concurrency Control and Implications in Distributed Databases," *Proc. COMPSAC 79 Conf.*, Chicago, Nov. 1979.
- [BEHR] Bayer, R., E. Elhardt, H. Heller, and A. Reiser. "Distributed Concurrency Control in Database Systems," *Proc. Sixth Int. Conf. on Very Large Data Bases*, IEEE, N.Y., 1980, pp. 275-284.
- [BHR] Bayer, R., H. Heller, and A. Reiser. "Parallelism and Recovery in Database Systems," ACM Trans. on Database Sys. 5, 2 (June 1980), pp. 139-156.
- [BG1] Bernstein, P.A. and N. Goodman, "Concurrency Control in Distributed Database Systems," Computing Surveys 13, 2 (June 1981), pp. 185-221.
- [BG2] Bernstein, P.A. and N. Goodman. "Concurrency Control Algorithms for Multiversion Database Systems," submitted for publication.
- [BGL] Bernstein, P.A., N. Goodman, and M.Y. Lai. "A Two-Part Proof Schema for Database Concurrency Control," Proc. 1981 Berkeley Workshop on Distributed Databases and Computer Networks.
- [BRGP] Bernstein, P.A., Rothnie, J.B., Goodman, N., and Papadimitriou, C.H.
 "The Concurrency Control Mechanism of SDD-1: A System for
 Distributed Databases (The Fully Redundant Case)," IEEE Trans. on
 Software Engineering, Vol. SE-4, No. 3 (May 1978).
- [BS2] Bernstein, P.A. and Shipman, D. "The Correctness of Concurrency Mechanisms in a System for Distributed Databases (SDD-1)," ACM Trans. on Database Systems, Vol. 5, No. 1, March 1980.
- [BSR] Bernstein, P.A., Shipman, D., and Rothnie, J. "Concurrency Control in a System for Distributed Databases (SDD-1)," ACM Trans. on Database Systems, Vol. 5, No. 1, March 1980.

- [BSW] Bernstein, P.A., Shipman D.W., and Wong, W.S. "Formal Aspects of Serializability in Database Concurrency Control," *IEEE Trans. on Software Engineering*, Vol. SE-5, No. 3, May 1979.
- [Casa] Casanova, M.A. The Concurrency Control Problem of Database
 Systems, Lecture Notes in Computer Science, Vol. 116, SpringerVerlag, 1981 (originally published as TR-17-79, Center for Research
 in Computing Technology, Harvard University, 1979).
- [CB] Cheng, W.K., and G.C. Belford. "Update Synchronization in Distributed Databases," Proc. 6th Int. Conf. on Very Large Data Bases, Oct. 1980.
- [Dubo] DuBourdieu, D.J., "Implementation of Distributed Transactions," Proc. 1982 Berkeley Workshop on Distributed Data Management and Computer Networks, pp. 81-94.
- [Ellis] Ellis, C.A. "A Robust Algorithm for Updating Duplicate Databases," Proc. 2nd Berkeley Workshop on Distributed Databases and Computer Networks, May 1977.
- [EGLT] Eswaran, K.P., Gray, J.N., Lorie, R.A., and Traiger, I.L. "The Notions of Consistency and Predicate Locks in a Database Systems," Communications of the ACM, Vol. 19, No. 11, November 1976.
- [G-M1] Garcia-Molina, H. "Performance Comparisons of Two Update Algorithms for Distributed Databases," Proc. 3rd Berkeley Workshop on Distributed Databases and Computer Networks, August 1978.
- [G-M2] Garcia-Molina, H. "Performance of Update Algorithms for Replicated Data in a Distributed Database," Ph.D. Dissertation, Computer Science Department, Stanford University, June 1979.
- [G-M3] Garcia-Molina, H. Stanford University, Stanford, California. "A Concurrency Control Mechanism for Distributed Data Bases Which Uses Centralized Locking Controllers," Proc. 4th Berkeley Conf. on Distributed Data Management & Computer Networks, August 1979.
- [GS] Gelenbe, E. and Sevcik, K. "Analysis of Update Synchronization for Multiple Copy Databases," *Proc. 3rd Berkeley Workshop on Distributed Databases and Computer Networks*, August 1978.
- [GISh] Gligor, V.D. and S.H. Shattuck, "On Deadlock Detection in Distributed Systems," *IEEE Trans. on Software Engineering*, Vol. SE-6, No. 5, September 1980, pp. 435-440.
- [Giff] Gifford, D.K. "Weighted Voting for Replicated Data," Proc. 7th Symp. on Operating Sys. Principles, ACM, N.Y., Dec. 1979.

- [Gray] Gray, J.N.. "Notes on Database Operating Systems," Operating Systems: An Advanced Course, Vol. 60, Lecture Notes in Computer Science, Springer-Verlag, N.Y., 1978, pp. 393-481.
- [GLPT] Gray, J.N., Lorie, R.A., Putzulo, G.R., and Traiger, I.L.
 "Granularity of Locks and Degrees of Consistency in a Shared
 Database," IBM Research Report RJ1654, September 1975.
- [GMBL] Gray, J.N., P. McJones, M. Blasgen, B. Lindsay, R. Lorie, T. Price, F. Putzolu, I. Traiger. "The Recovery Manager of the System R Database Manager," Computing Surveys 13, 2 (June 1981), pp. 223-242.
- [HS] Hammer, M. and D.W. Shipman. "Reliability Mechanisms for SDD-1: A System for Distributed Databases," ACM Trans. on Database Sys. 5, 4 (Dec. 1980), pp. 431-466.
- [Holt] Holt, R.C., "Some Deadlock Properties of Computer Systems," Computing Surveys 4, 3 (Dec. 1972), pp. 179-195.
- [KNTH] Kaneko, A., Y. Nishihara, K. Tsuruoka, and M. Hattori. "Logical Clock Synchronization Method for Duplicated Database Control," Proc. First International Conf. on Distributed Computing Systems, IEEE, N.Y., Oct. 1979, pp. 601-611.
- [KMIT] Kawazu, S., Minami, S., Itoh, K., and Teranaka, K. "Two-Phase Deadlock Detection Algorithm in Distributed Databases," *Proc.* 1979 International Conference on Very Large Data Bases, IEEE, N.Y.
- [KC] King, P.F., and Collmeyer, A.J. "Database Sharing--An Efficient Mechanism for Supporting Concurrent Processes," *Proc.* 1974 NCC, AFIPS Press, Montvale, New Jersey, 1974.
- [KR] Kung, H.T. and Robinson, J.T. "On Optimistic Methods for Concurrency Control," *Proc. 1979 Int. Conf. on Very Large Data Bases*, Oct. 1979.
- [Lamp] Lamport, L. "Time, Clocks, and the Ordering of Events in a Distributed System," Comm. of the ACM 21, 7 (July 1978), pp. 558-565.
- [LS] Lampson, B. and Sturgis, H. "Crash Recovery in a Distributed Data Storage System," Tech. Report, Computer Science Laboratory, Xerox, Palo Alto Research Center, Palo Alto, Calif. 1976.
- [Lelann] LeLann, G. "Algorithms for Distributed Data-Sharing Systems Which Use Tickets," Proc. 3rd Berkeley Workshop on Distributed Databases and Computer Networks, August 1978.
- [Lin] Lin, W.K. "Concurrency Control in a Multiple Copy Distributed Data Base System," Proc. 4th Berkeley Conference on Distributed Data Management & Computer Networks, August 1979.

- [LN] Lin, W.T.K. and J. Nolte. "Performance of Two Phase Locking,"

 Proc. 1982 Berkeley Workshop on Distributed Data Management and

 Computer Networks, pp. 131-160.
- [MM] Menasce, D.A. and Muntz, R.R. "Locking and Deadlock Detection in Distributed Databases," *IEEE Transactions on Software Engineering*, Vol. SE-5, No. 3, May 1979, pp. 195-202.
- [MPM] Menasce, D.A., G.J. Popek and R.R. Muntz. "A Locking Protocol for Resource Coordination in Distributed Databases," *Proc.* 1978 ACM-SIGMOD Conf. on Management of Data, ACM, N.Y.
- [Mino] Minoura, T. "A New Concurrency Control Algorithm for Distributed Data Base Systems," Proc. 4th Berkeley Conference on Distributed Data Management & Computer Networks, August 1979.
- [Montgomery] Montgomery, W.A. "Robust Concurrency Control for a Distributed Information System," Ph.D. Dissertation, Laboratory for Computer Science, MIT, Dec. 1978.
- [PBR] Papdimitriou, C.H., Bernstein, P.A., and Rothnie, J.B., Jr.
 "Some Computational Problems Related to Database Concurrency
 Control," Proc. Conf. on Theoretical Computer Science, Waterloo,
 Ontario, August, 1977.
- [Papadimitriou] Papadimitriou, C.H. "Serializability of Concurrent Updates," Journal of the ACM, Vol. 26, No. 4, Oct. 1979, pp. 631-653.
- [Reed] Reed, D.P. "Naming and Synchronization a Decentralized Computer System," Ph.D. Thesis, MIT Department of Electrical Engineering, Sept. 1978.
- [Riesl] Ries, D. "The Effect of Concurrency Control on Database Management System Performance," Ph.D. Dissertation, Computer Science Department, University of California, Berkeley, April 1979.
- [Ries2] Ries, D. "The Effects of Concurrency Control on the Performance of a Distributed Data Management Systems," Proc. 4th Berkeley Conf. on Distributed Data Management & Computer Networks, August 1979.
- [RSL] Rosenkrantz, D.J., Stearns, R.E., and Lewis, P.M. "System Level Concurrency Control for Distributed Database Systems," ACM Trans. on Database Systems, Vol. 3, No. 2 (June 1978), pp. 178-198.
- [SM] Shapiro, R.M. and Millstein, R.E. "Reliability and Fault Recovery in Distributed Processing," Oceans '77 Conference Record, Vol. II, Los Angeles, 1977.
- [SK] Silberschatz, A. and Z. Kedem, "Consistency in Hierarchical Database Systems," *Journal of the ACM*, Vol. 27, No. 1, Jan. 1980, pp. 72-80.

- [SRL] Stearns, R.E., Lewis, P.M., II, and Rosenkrantz, D.J. "Concurrency Controls for Database Systems," Proc. of the 17th Annual Symposium on Foundations of Computer Science, IEEE, 1976, pp. 19-32.
- [SR] Stearns, R.E., and D.J. Rosenkrantz. "Distributed Database Concurrency Controls Using Before-Values," *Proc. 1981 ACM-SIGMOD Conf.*, ACM, N.Y., pp. 74-83.
- [Stonebraker] Stonebraker, M. "Concurrency Control and Consistency of Multiple Copies of Data in Distributed INGRES," *IEEE Transactions on Software Engineering*, Vol. SE-5, No. 3, May 1979, pp. 188-194.
- [Thom] Thomas, R.H. "A Majority Consensus Approach to Concurrency Control for Multiple Copy Databases," ACM Trans. on Database Systems, Vol. 4, No. 2, June 1979, pp. 180-209.

SECTION III

MULTIVERSION CONCURRENCY CONTROL

-- THEORY AND ALGORITHMS*

Philip A. Bernstein

Nathan Goodman

^{*}An extended abstract of this paper appeared in the Proceedings of the First ACM SIGACT-SIGOPS Symposium on Principles of Distributed Computing. The complete paper has been accepted for publication in ACM Transactions on Database Systems.

ABSTRACT

Concurrency control is the activity of synchronizing operations issued by concurrently executing programs on a shared database. The goal is to produce an execution that has the same effect as a serial (noninterleaved) one.

In a multiversion database system, each write on a data item produces a new copy (or *version*) of that data item. This paper presents a theory for analyzing the correctness of concurrency control algorithms for multiversion database systems. We use the theory to analyze some new algorithms and some previously published ones.

1. INTRODUCTION

A database system (dbs) is a process that executes read and write operations on data items of a database. A transaction is a program that issues reads and writes to a dbs. When transactions execute concurrently, the interleaved execution of their reads and writes by the dbs can produce undesirable results. Concurrency control is the activity of avoiding such undesirable results. Specifically, the goal of concurrency control is to produce an execution that has the same effect as a serial (noninterleaved) one. Such executions are called serializable.

A dbs attains a serializable execution by controlling the order in which reads and writes are executed. When an operation is submitted to the dbs, the dbs can either execute the operation immediately, delay the operation for later processing, or reject the operation. If an operation is rejected, then the transaction that issued the operation is aborted, meaning that all of the transaction's writes are undone, and transactions that read any of the values produced by those writes are also aborted.

The principal reason for rejecting an operation is that it arrived "too late". For example, a read is normally rejected because the value it was supposed to read has already been overwritten. Such rejections can be avoided by keeping old copies of each data item. Then a tardy read can be given an old value of a data item, even though it was "overwritten".

In a multiversion dbs, each write on a data item x, say, produces a new copy (or version) of x. For each read on x, the dbs selects one of the versions of x to be read. Since writes do not overwrite each other, and since reads can read any version, the dbs has more flexibility in controlling the order of reads and writes. Several interesting concurrency

control algorithms that exploit multiversions have been proposed [BEHR,BHR, CFLNR,Dubo,Reed,Silb, SLR,SR]. Theoretical work on this problem includes [PK,SLR].

This paper presents a theory for analyzing the correctness of concurrency control algorithms for multiversions dbs's. We present some new multiversion algorithms. We use the theory to analyze the new algorithms and several previously published ones.

Section 2 reviews concurrency control theory for non-multiversion databases. Section 3 extends the theory to multiversion databases.

Sections 4-6 use the theory to analyze multiversion concurrency control algorithms.

2. BASIC SERIALIZABILITY THEORY

The standard theory for analyzing database concurrency control algorithms is serializability theory [BSW.Casa,EGLT,Papa,PBR,SLR]. Serializability theory is a method for analyzing executions allowed by the concurrency control algorithm. The theory gives a precise condition under which an execution is correct. A concurrency control algorithm is then judged to be correct if all of its executions are correct.

This section reviews serializability theory for concurrency control without multiversions.

2.1 System Model

We assume the dbs is distributed and use Lamport's model of distributed executions [Lamp]. The system consists of a collection of processes that communicate by passing messages. The model describes an execution in terms of a happens before relation that tells the order in which events occur. An event is one of the following: the execution of an operation by a process, the sending of a message, or the receipt of a message.

Within a process, the happens before relation is any partial order over the process's events. For the system, the happens before relation (denoted <) is the *smallest* partial order over all events in the system such that: (1) If e and f are events in process P, and e happens before f in P, then e < f. (2) If e is the event "Process P sends message M" and f is the event "Process Q receives M'", then e < f. Condition (1) states that < must be consistent with the order of events within each process. Condition (2) states that a message must be sent

before it is received. And, since < is the smallest partial order satisfying these conditions, condition (2) is the only way that events in different processes can be ordered.

This paper deals at a higher level of abstraction. Hereafter, we will not explicitly mention processes and messages (except briefly in Section 6).

For concreteness, the reader may assume that each transaction is a process, and each data item is managed by a separate process. (Our results don't depend on these assumptions.) Under these assumptions each database operation entails two message exchanges. For transaction T_i to read x, T_i must send a message to x's process; to return x's value, the x process must send a message to T_i . The same message pattern is needed for writes; in this case, the return message just acknowledges that the write has been done. Also under these assumptions, any decision or event ordering involving one data item is a local activity; decisions or orderings involving multiple data items are distributed activities. The abstraction that we use hides message exchanges and related issues, allowing us to reason about concurrency control at a higher level.

2.2 Logs

Serializability theory models executions by *logs*. A log identifies the Read and Write operations executed on behalf of each transaction, and tells the order in which those operations were executed. A log is an abstraction of Lamport's happens before relation.

A transaction log represents an allowable execution of a single transaction. Pormally, a transaction log is a partially ordered set (poset) $T_i = (\Sigma_i, <_i)$ where Σ_i is the set of Reads and Writes issued by (an execution of) transaction i, and <_i tells the order in which those operations must be executed. We write transaction logs as diagrams.

$$T_{1} = \begin{cases} r_{1}(x) \\ w_{1}(z) \end{cases}$$

 T_1 represents a transaction that reads x and z in parallel, and then writes x. (Presumably, the value written depends on the values read.)

We use $\mathbf{r_i}[\mathbf{x}]$ (resp., $\mathbf{w_i}[\mathbf{x}]$) to denote a Read (resp., Write) on \mathbf{x} issued by $\mathbf{T_i}$. To keep this notation unambiguous, we assume that no transaction reads or writes a data item more than once. None of our results depend on this assumption.

Let $T = \{T_0, ..., T_n\}$ be a set of transaction logs. A ds log (or simply a log) over T represents an execution of $T_0, ..., T_n$. Formally, a log over T is a poset $L = (\Sigma, <)$ where

- 1. $\Sigma = \bigcup_{i=0}^{n} \Sigma_{i};$
- 2. $\leq y_{i=0}^{n} < ;$
- 3. every $r_j[x]$ is preceded by at least one $w_j[x]$ (i = j is possible), where $w_i[x]$ precedes $r_j[x]$ is synonymous with $w_j[x] < r_j[x]$; and
- 4. all pairs of conflicting operations are < related (two operations conflict if they operate on the same data item, and at least one is a Write).</p>

Condition (1) states that the dbs executed all, and only, those operations submitted by T_0, \ldots, T_n . Condition (2) states that the dbs honored all operation orderings stipulated by the transactions. Condition (3) states that no transaction can read a data item until some transaction has written

its initial value. Condition (4) states that the dbs executes conflicting operations, sequentially. E.g., if T_i reads x and T_j writes x, $r_i[x]$ happens before $w_j[x]$ or vice versa; the operations cannot occur at the same time.

Consider the following transaction logs

$$T_0 = w_0[x]$$
 $w_0[y]$
 $w_0[z]$
 $T_1 = r_1[x]$
 $r_1[x]$

The following are some of the possible logs over $\{T_0, T_1\}$

(1)
$$\begin{array}{c} w_0[x] \longrightarrow r_1[x] \\ w_0[y] & \downarrow \\ w_0[z] \longrightarrow r_1[z] \end{array}$$

(2)
$$\begin{array}{c} w_0[x] \longrightarrow r_1[x] \\ w_0[y] & \downarrow \\ w_0[z] \longrightarrow r_1[z] \end{array}$$

(3)
$$\begin{array}{c} w_0[x] \rightarrow r_1[x] \\ w_0[y] \\ w_0[z] \rightarrow r_1[z] \end{array}$$

Note that orderings implied by transitivity are usually not drawn. E.g. $w_0[x] < w_1[x]$ is not drawn in the diagrams, although it follows from $w_0[x] < r_1[x] < w_1[x]$,

Notice that the dbs is allowed to process $\operatorname{Read}(x)$ and $\operatorname{Read}(z)$ sequentially (cf. (1) and (2)), even though T_1 allows them to run in parallel. However, the dbs is not allowed to reverse or eliminate any ordering stipulated by T_1 .

Given transaction logs

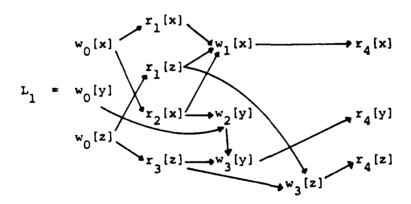
$$T_{0} = w_{0}[y] \qquad T_{1} = w_{1}[x]$$

$$W_{0}[z] \qquad T_{1} = w_{1}[x]$$

$$T_{2} = r_{2}[x] + w_{2}[y] \qquad T_{3} = r_{3}[z] \qquad w_{3}[y] \qquad T_{4} = r_{4}[y]$$

$$W_{3}[z] \qquad T_{4} = r_{4}[y]$$

the following is a log over $\{T_0, T_1, T_2, T_3, T_4\}$.



The following is another log over the same transactions. $L_2 = w_0[x]w_0[y]w_0[z]r_2[x]w_2[y]r_1[x]r_1[z]w_1[x]r_3[z]w_3[y]w_3[z]r_4[x]r_4[y]r_4[z].$

When we write a log as a sequence, e.g. L_2 , we mean that the log is totally ordered: each operation precedes the next one and all subsequent ones in the sequence. Thus, in L_2 , $w_0[x] < w_0[y] < w_0[z] < r_2[x]$

2.3 Log Equivalence

Intuitively, two logs are equivalent if each transaction performs the same computation in both logs. We formalize log equivalence in terms of information flow between transactions.

Let L be a log over $\{T_0, \ldots, T_n\}$. Transaction T_j reads-x-from T_i in L if (1) $w_i[x]$ and $r_j[x]$ are operations in L; (2) $w_i[x] < r_j[x]$; and (3) no $w_i[x]$ falls between these operations. Two logs over $\{T_0, \ldots, T_n\}$ are equivalent, denoted Ξ , if they have the same reads-from's; i.e. for all i,j, and x, T_j reads-x-from T_i in one log iff this condition holds in the other. This definition ensures that each transaction reads the same values from the database in both logs.

Consider logs L_1 and L_2 of the previous section. These logs have the same reads-from's:

 T_1 reads-x-from T_0 , T_1 reads-z-from T_0

 T_2 reads-x-from T_0

 T_3 reads-z-from T_0

 $\mathbf{T_4} \quad \text{reads-x-from} \quad \mathbf{T_1}, \quad \mathbf{T_4} \quad \text{reads-y-from} \quad \mathbf{T_3}, \quad \mathbf{T_4} \quad \text{reads-z-from} \quad \mathbf{T_3}.$ Therefore, $\mathbf{L_1} \stackrel{\scriptstyle \equiv}{=} \mathbf{L_2}.$

This definition of log equivalence ignores the final database state produced by the logs. For example, these logs are equivalent

 $L = w_0(x) w_1(x)$

 $L' = w_1[x] w_0[x]$

even though different transactions produce the final value of x in each log. It is often desirable to strengthen the notion of equivalence by insisting that for each x, the same transaction writes the final value of x in both logs. This can be modelled by (i) adding a "final transaction" that follows all other transactions and reads the entire database (e.g., T_4 in logs L_1 and L_2); and (ii) redefining equivalence to be that the logs have the same reads-from's and the same final transaction.

2.4 Serializable Logs

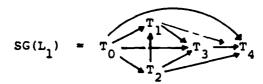
A serial log is a totally ordered log on I such that for every pair of transactions T_i and T_j, either all of T_i's operations precede all of T_j's, or vice versa (e.g., L₂ in Section 2.2). A serial log represents an execution in which there is no concurrency whatsoever; each transaction executes from beginning to end before the next transaction begins. From the point of view of concurrency control, therefore, every serial log represents an obviously correct execution.

What other logs represent correct executions? From the point of view of concurrency control, a correct execution is one in which concurrency is invisible. That is, an execution is correct if it is equivalent to an execution in which there is no concurrency. Serial logs represent the latter executions, and so a correct log is any log equivalent to a serial log. Such logs are termed serializable (SR).

Log L_1 of Sec. 2.2 is SR, because it is equivalent to serial log L_2 of Sec. 2.3. Therefore L_1 is a correct log.

2.5 The Serializability Theorem

Let L be a log over $\{T_0, \ldots, T_n\}$. The serialization graph for L, SG(L), is a directed graph whose nodes are T_0, \ldots, T_n , and whose edges are all $T_i + T_j$ (i\neq j) such that some operation of T_i precedes and conflicts with some operation of T_j . The serialization graph for example log L₁ is



Edge $T_0 \to T_1$ is present because $w_0[x] < r_1[x]$. Edge $T_1 \to T_3$ is present because $r_1[z] < w_3[z]$. And so forth.

SERIALIZABILITY THEOREM [BSW,EGLT,Papa,PBR,SLR]. If SG(L) is acyclic then L is SR.

3. MULTIVERSION SERIALIZABILITY THEORY

In a multiversion dbs, each write produces a new version. We denote versions of x by x_i , x_j ..., where the subscript is the index of the transaction that wrote the version. Operations on versions are denoted $r_i[x_j]$ and $w_i[x_i]$.

3.1 Multiversion Logs

been produced.

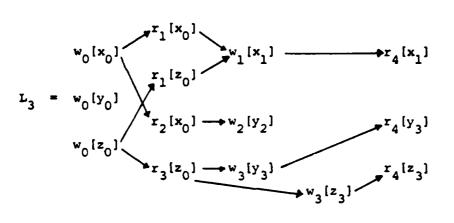
Let $T = \{T_0, \dots, T_n\}$ be a set of transaction logs (defined exactly as in Section 2.2, i.e., the operations reference data items). To execute T, a multiversion dbs must translate T's "data item operations" into "version operations". We formalize this translation by a function T which maps each T into T i

A multiversion dbs log (or simply mv log) over T is a poset $L = (\Sigma, <)$ where

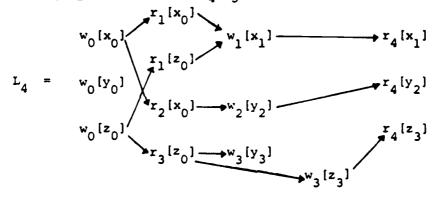
- 1. $\Sigma = h(U_{i=1}^{n} \Sigma_{i})$, for some translation function h,
- 2. for each T_i , and all operations op and op, if $o_i < o_i$, if $o_i < o_i$, then $h(op_i) < h(op_i)$, and
- 3. if $h(r_j[x]) = r_j[x_i]$, then $w_i[x_i] < r_j[x_i]$.

 Condition (1) states that each operation submitted by a transaction is translated into an appropriate multiversion operation. Condition (2) states that the mv log preserves all orderings stipulated by transactions. Condition (3) states that a transaction may not read a version until it's

The following is an mv log over $\{T_0, T_1, T_2, T_3, T_4\}$ of Section 2.



All mv logs over a set T have the same write operations, since $h(w_i[x]) = w_i[x_i].$ But they needn't have the same reads. For example, $L_4 \quad \text{has} \quad r_4[y_2] \quad \text{instead of} \quad r_4[y_3].$



3.2 MV Log Equivalence

Most definitions and results from basic serializability theory extend to mv logs: we simply replace the notion of "data item" by "version" in those definitions and results. However, the structure of mv logs simplifies the treatment. This section redoes the material of Sections 2.3 and 2.4 for mv logs.

Let I, be an mv log over $\{T_0, \dots, T_n\}$. Transaction T_j reads-x-from T_i in L if T_j reads the version x produced by T_i . By definition, the version of x produced by T_i is x_i . So, T_j reads-x-from T_i iff

T reads x_i. This means that the reads-from's in L are determined by the translation function h, viz. the way h translates "data item reads" into "version reads".

Two mv logs over $\{T_0, \ldots, T_n\}$ are equivalent, denoted Ξ , if they have the same reads-from's. The reads-from's in an mv log are determined by its read operations: T_j reads-x-from T_i iff $r_j[x_i]$ is an operation of the log. So, two logs are equivalent iff they have the same read operations. Moreover, since all mv logs over the same transactions have the same writes, equivalence reduces to a trivial condition.

FACT 1. Two mu logs over a set of transactions T are equivalent iff the logs have the same operations.

Two "version operations" conflict if they operate on the same version and one is a write. Only one pattern of conflict is possible in an mv log: If $\operatorname{op}_i < \operatorname{op}_j'$ and these operations conflict, then op_i is $\operatorname{w}_i[x_i]$ and op_j is $\operatorname{r}_j[x_i]$. Conflicts of the form $\operatorname{w}_i[x_i] < \operatorname{w}_j[x_i]$ are impossible, because each write produces a new version. Conflicts of the form $\operatorname{r}_j[x_i] < \operatorname{w}_i[x_i]$ are impossible since T_j can't read x_i until it's been produced. Thus all conflicts in an mv log correspond to reads-from's.

The serialization graph for an mv log is defined as for a regular log. Since conflicts are so structured in an mv log, serialization graphs are quite simple. Let L be an mv log over $\{T_0, \ldots, T_n\}$. SG(L) has nodes T_0, \ldots, T_n and edges $T_i \to T_j$ (i \neq j) such that for some x, T_j reads-x-from T_i . That is, $T \to T_j$ is present iff for some x, $T_j[x_i]$ is an operation of L. This gives us the following.

FACT 2. Let L and L' be mu logs over T.

- (i) If L and L' have the same operations, then SG(L) = SG(L').
- (ii) If L and L' are equivalent, then SG(L) = SG(L').

The serialization graphs for logs L_3 and L_4 of the previous section are given below.

$$SG(L_3) = T_0 \xrightarrow{T_1} T_3$$

$$SG(L_4) = T_0$$

$$T_2$$

(Compare to $SG(L_1)$ in Section 2.4.)

3.3 One-Copy Serializability

Although the database has multiple versions, users expect their transactions to behave as if there were just one copy of each data item. Serial logs don't always behave this way. Here is a simple example.

$$w_0[x_0]w_0[y_0]r_1[x_0]w_1[y_1]r_2[y_0]w_2[x_2]$$
.

 T_2 reads-y-from T_0 even though T_1 comes between T_0 and T_2 and produces a new value for y. This behavior cannot be reproduced with only one copy of y. In a one copy database, if T_0 comes before T_1 and T_1 is before T_2 , then T_2 must read the value of y produced by T_1 .

We must therefore restrict the set of allowable serial logs.

A serial mv log L is one-copy serial (or 1-serial) if for all i,j, and x, if T_j reads-x-from T_i then i=j or T_i is the last transaction preceding T_j that writes into any version of x. (Since L is totally ordered, the word "last" in this definition is well-defined.) The log above is not 1-serial, because T_2 reads-y-from T_0 , but $w_0\{y_0\} < w_1\{y_1\} < r_2\{y_0\}$. L_5 below is 1-serial.

A log is one-copy serializable (or 1-SR) if it's equivalent to a 1-serial log. For example, L_3 of Section 3.1 is equivalent to L_5 , as can be verified by Fact 1; hence L_3 is 1-SR. L_4 is equivalent to no 1-serial log (this can be verified by checking all possible serial logs with the same operations as L_4); hence L_4 is not 1-SR.

It is possible for a serial log to be 1-SR even though it is not 1-serial itself. For example

$$w_0[x_0] r_1[x_0] w_1[x_1] r_2[x_0]$$

is not 1-serial since T_2 reads-x-from T_0 instead of T_1 . But it is 1-SR, because it is equivalent to

$$w_0[x_0] r_2[x_0] r_1[x_0] w_1[x_1]$$
.

One-copy serializability is our correctness criterion for multiversion concurrency control. The following theorem justifies this criterion, proving that an mv log behaves like a serial non-mv log iff the mv log is 1-SR.

First, we extend our notion of log equivalence to handle mv and non-mv logs. Let L and L' be (mv or non-mv) logs over T. L and L' are equivalent, E, if they have the same reads-from's.

1-SR EQUIVALENCE THEOREM. Let L be an mv log over T. L is equivalent to a serial, non-mv log over T iff L is 1-SR.

Proof (if). Let L_s be a 1-serial log equivalent to L. Form a serial, non-mv log L_s' by translating each $w_i[x_i]$ into $w_i[x]$ and $r_j[x_i]$ into $r_j[x]$. Consider any reads-from in L_s , say T_j reads-x-from T_i . Since L_s is 1-serial, no $w_k[x_k]$ lies between $w_i[x_i]$ and $r_j[x_i]$. Hence no $w_k[x]$ lies between $w_i[x]$ and $r_j[x]$ in L_s' . Thus, T_j reads-x-from T_i in L_s' . This establishes $L_s' \in L_s$. Since $L_s \in L_s$, $L \in L_s'$ follows by transitivity (since i is an equivalence relation).

(only if). Let L_s^* be the hypothesized serial, non-mv log equivalent to L. Translate L_s^* into a serial mv log L_s by mapping each $v_i[x]$ into $w_i[x_i]$ and each $r_j[x]$ into $r_j[x_i]$ such that T_j reads-x-from T_i in L_s^* . This translation preserves reads-from's, so $L_s \equiv L_s^*$. By transitivity, $L \equiv L_s$.

It remains to prove that L_s is 1-serial. Consider any reads-from in L_s' , say T_j reads-x-from T_i . Since L_s' is a non-mv log, no $w_k[x]$ lies between $w_i[x]$ and $r_j[x]$. Hence no $w_k[x_k]$ lies between $w_i[x_i]$ and $r_j[x_i]$ in L_s . Thus, L_s is 1-serial, as desired.

3.4 The 1-Serializability Theorem

To tell if an mv log is 1-SR we use a modified serialization graph.

Given a log L and data item x, a version order for x is any (non-reflexive) total order overall of x's versions written in L. A version order, <<, for L is the union of the version orders for all data items. A possible version order for L₃ of Section 3.1 (or L₅ of Section 3.3) is

Given L and a version order <<, the multiversion serialization graph, MVSG(L,<<), is SG(L) with the following edges added:

(*) for each $r_k[x_j]$ and $w_i[x_i]$ in L, $k \neq i$, if $x_i << x_j$ then include $T_i \rightarrow T_j$, else include $T_k \rightarrow T_i$.

For example,

$$MVSG(L_3, <<) = T_0 \xrightarrow{T_1} T_4$$

(Compare to $SG(L_1)$ in Section 2.5.)

The following theorem is our principal tool for analyzing multiversion concurrency control algorithms.

1-SERIALIZABILITY THEOREM. An mv log L is 1-SR iff there exists a version order << such that MVSG(L,<<) is acyclic.

Proof (if). Let L_s be a serial mv log induced by a topological sort of MVSG(L,<<). I.e., L_s is formed by topologically sorting MVSG(L,<<), and as each node T_i is listed in the sort. T_i 's operations in L are added to L_s one by one in any order consistent with L. L_s has the same operations as L, so by Fact 1, $L \equiv L_s$.

It remains to prove that L_s is 1-serial. Consider any reads-from situation say T_k reads-x-from T_j . Let $w_i\{x_i\}$ be any other write on a

version of x. If $x_i << x_j$, then by rule (*) of the MVSG definition, the graph includes $T_i + T_j$. This edge forces T_j to follow T_i in L_s . If $x_j << x_i$, then by rule (*), MVSG(L,<<) includes $T_k + T_i$. This forces T_k to precede T_i in L_s . In both cases, T_i is prevented from falling between T_j and T_k . Since T_i was an arbitrary writer on x, this proves that no transaction that writes a version of x comes between T_j and T_k in L_s . Thus L_s is 1-serial.

(only if) Given L and <<, let MV(L,<<) be the graph specified by statement (*) of the MVSG definition. Statement (*) depends only on the operations in L and <<; it does not depend on the order of operations in L. Thus, if L_1 and L_2 are multiversion logs with the same operations, then MV(L_1 ,<<) = MV(L_2 ,<<), for all version orders <<.

Let L_s be a 1-serial log equivalent to L. All edges in $SG(L_s)$ go "left-to-right", i.e. if $T_i \to T_j$ then T_i is before T_j in L_s . Define << by: $x_i << x_j$ only if T_i is before T_j in L_s . All edges in $MV(L_s, <<)$ are also left-to-right. Therefore all edges in $MVSG(L_s, <<) = MV(L_s, <<) \cup SG(L_s)$ are left-to-right, too. This implies that $MVSG(L_s, <<)$ is acyclic.

By Fact 1, L and L_s have the same operations. Hence, $MV(L,<<) = MV(L_s,<<) \, . \quad \text{By Fact 2, SG}(L) = SG(L_s) \, . \quad \text{Therefore } MVSG(L,<<) = \\ MVSG(L_s,<<) \, . \quad \text{Since } MVSG(L_s,<<) \, \text{ is acyclic, so is } MVSG(L,<<) \, .$

Sections 4-6 use the 1-Serializability Theorem to analyze multiversion concurrency control algorithms. We conclude this section with a complexity result.

3.5 1-Serializability is NP-Complete

1-SR COMPLEXITY THEOREM. It is NF-complete to decide whether an mu log is 1-SR.

Proof (membership in NP). Let L be an mv log over T. Guess a 1-serial log L over T and verify L \equiv L. By Fact 1, we can verify L \equiv L by comparing the logs' operation sets.

(NP-hardness). The reduction is from the log SR problem. Let L' be a non-mv log over T. Map L' into an equivalent mv log L by translating each $w_i[x]$ into $w_i[x_i]$ and each $r_j[x]$ into $r_j[x_i]$ such that T_j reads-x-from T_i in L'. By the 1-SR Equivalence Theorem, L is 1-SR iff there exists a non-mv serial log L' such that $L \equiv L'_s$. But, by transitivity, L' exists iff L' is SR. Thus L' is SR iff L is 1-SR.

Papadimitriou and Kanellakis prove that a related problem is NP-complete [PK]: Given a conventional log L, can one transform L into a 1-SR mv log by mapping each $w_i[x]$ into $w_i[x_i]$ and each $r_j[x]$ into $r_j[x_i]$ for some x_i where $w_i[x] < r_j[x]$? This problem corresponds to choosing versions for reading after having scheduled the operations. Our problem corresponds to choosing versions at the same time as scheduling the operations.

4. MULTIVERSION TIMESTAMPING

The earliest multiversion concurrency control algorithm that we know of is Reed's multiversion timestamping algorithm [Reed].

Each transaction, T_i, is assigned a unique *timestamp*, TS(i), when it begins executing. Intuitively, the timestamp tells the "time" at which the transaction began. Formally, timestamps are just numbers with the property that each transaction is assigned a different timestamp. Each Read and Write carries the timestamp of the transaction that issued it, and each version carries the timestamp of the transaction that wrote it.

Operations are processed first-come-first-served. But the translation from data item operations to version operations makes it appear as if operations were processed in timestamp order.

The algorithm works as follows.

- •r_i[x] is translated into $r_i[x_k]$, where x_k is the version of x with largest timestamp \leq TS(i).
- • \mathbf{w}_{i} [x] has two cases. If the dbs has already processed \mathbf{r}_{j} [x] such that TS(k) < TS(i) < TS(j), then \mathbf{w}_{i} [x] is rejected. Otherwise \mathbf{w}_{i} [x] is translated into \mathbf{w}_{i} [x]. Intuitively, \mathbf{w}_{i} [x] is rejected if it would invalidate \mathbf{r}_{i} [x].

We wish to use serializability theory to prove this algorithm correct. To do so, we must state the algorithm in serializability theoretic terms. We take the description of the algorithm above and infer properties that all logs produced by the algorithm will satisfy. These properties form our formal definition of the algorithm. We use serializability theory to prove that these Yog properties imply 1-serializability.

The following properties form our formal definition of the mv time-stamping algorithm. Let L be an mv log over $\{T_0,\ldots,T_n\}$.

- TS1. Every T_i has a numeric timestamp TS(i) satisfying a uniqueness condition: TS(i) = TS(j) iff i = j.
- TS2. Every $r_k[x_j]$ and $w_i[x_i]$ are < related; i.e., $r_k[x_j] < w_i[x_i]$ or vice versa.
- TS3.1 For every $r_k[x_j]$, TS(j) \leq TS(k).
- TS3.2 For every $r_k[x_j]$ and $w_i[x_i]$, $i \neq j$, if $w_i[x_i] < r_k[x_j]$, then either TS(i) < TS(j) or TS(k) \leq TS(i).
- TS4. For every $r_k[x_j]$ and $w_i[x_i]$, $i \neq j$, if $r_k[x_j] < w_i[x_i]$, then either TS(i) < TS(j) or TS(k) \leq TS(i).

Property TS1 just says that transactions have unique timestamps. TS2 is implicit in the description of how the algorithm works; without this property, the statement, "If the dbs has already processed $r_j[x_k]...$ " is not well-defined. TS3 states that at the time $r_k[x_j]$ is processed, x_j is the version of x with largest timestamp \leq TS(k). TS4 states that once the dbs has processed $r_k[x_j]$, it will not process any $w_i[x_i]$ with TS(j) \leq TS(k).

Properties TS3.2 and TS4 can be simplified. By TS2, $r_k[x_j]$ and $w_i[x_i]$ are < related. So, TS3.2 and TS4 are equivalent to

TS5. For every $r_k[x_j]$ and $w_i[x_i]$, $i \neq j$, either TS(i) < TS(j) or TS(k) \leq TS(i).

We now prove that any log satisfying these properties is 1-SR. In other words, mv timestamping is a correct concurrency control algorithm.

MULTIVERSION TIMESTAMPING THEOREM. All logs produced by the mv timestamping algorithm are 1-SR.

<u>Proof.</u> Let L be a log produced by the algorithm. Define a version order by: $x_i << x_j$ implies TS(i) < TS(j). We prove that all edges in

MVSG(L,<<) are in timestamp order: if $T_i + T_j$ is an edge, then TS(i) < TS(j). Let $T_i + T_j$ be an edge of SG(L). This edge corresponds to a readsfrom situation, i.e., for some x, T_j reads-x-from T_i . By TS3.1 $TS(i) \le TS(j)$;
by TS1, $TS(i) \ne TS(j)$. So TS(i) < TS(j), as desired.

Consider any edge introduced by rule (*) of the MVSG definition. Let $w_i[x_i]$, $w_j[x_j]$, and $r_k[x_j]$ be the operations stipulated by rule (*). There are two cases. (1) $x_i << x_j$. Then the edge is $T_i + T_j$. TS(i) < TS(j) comes from our definition of <<. (2) $x_j << x_i$. Then the edge is $T_k + T_i$. By TS5, either TS(i) < TS(j) or $TS(k) \le TS(i)$. The first option is impossible, since the definition of << requires TS(j) < TS(i). By TS1, $TS(k) \ne TS(i)$. So, TS(k) < TS(i), as desired.

This proves that all edges in MVSG(L,<<) are in timestamp order. Since timestamps are numbers, hence totally ordered, it follows that MVSG(L,<<) is acyclic. So by the 1-Serializability Theorem, L is 1-SR.

5. MULTIVERSION LOCKING

Bayer et al. [BEHR, BHR] and Stearns and Rosenkrantz [SR] have presented multiversion algorithms that synchronize using a technique similar to locking. This section studies a generalization of their algorithms. As in the previous section, we start with an informal description of the algorithm. Then we state log properties the algorithm induces. Finally we prove that these log properties imply 1-serializability.

Each transaction and version exists in one of two states: certified or uncertified. When a transaction begins, it is uncertified; when a version is written, it, too, is uncertified. Later actions of the algorithm cause the transaction and all versions it wrote to become certified. The concept of certified corresponds to closed in [SR].

Let $c_i[x_i]$ be the event "x_i is certified." The algorithm requires that all $c_i[x_i]$ and $r_k[x_j]$ be < related. Also all $c_i[x_i]$ and $c_j[x_j]$ must be < related. A version order is defined by: $x_i << x_j$ iff $c_i[x_i] < c_j[x_j]$.

The algorithm works as follows.

 $r_i[x]$ is translated into $r_i[x_k]$ where x_k is either the *last* (w.r.t.<<) certified version of x, or *any* uncertified version. The algorithm may use any rule whatever for deciding which of these versions to read.

 $w_i[x]$ is translated into $w_i[x_i]$. As stated above, x_i is uncertified at this point.

•When a transaction finishes executing, the dbs attempts to certify it and all versions it wrote. For each data item x that T_i wrote, the dbs tries to set a certify-lock on x for T_i. This succeeds iff no other transaction already has a certify-lock on x; if the lock can't be set, T_i waits until it can. When T_i has all of its certify-locks, two further conditions must be satisfied:

- $\underline{C1}$. For each x_k that T_i read, $k \neq i$, x_k is certified.
- $\underline{C2}$. For each x_i that T_i wrote, and for each version x_k of x_i that is already certified, all transactions that read x_k have been certified.

Attaining Cl is just a matter of time; once Cl is satisfied no future event can cause it to become false. To attain C2, we set a certify-token on x to stop future reads from reading certified versions of x; instead, they may read x, or any other uncertified version of x.

When these conditions hold, T_i is declared to be certified. This fact is broadcast to all versions T_i wrote. When a version x_i receives this information, it, too, is certified; i.e., the event $c_i[x_i]$ occurs. When x_i is certified, the certify-lock and certify-token on x_i are released.

This algorithm, like most locking algorithms, can deadlock. Deadlocks can arise from two independent causes: (1) waiting for certify-locks; and (2) waiting for conditions C1 and C2. To detect deadlocks, the algorithm can use a directed blocking graph whose nodes are the transactions, and whose edges are all $T_i^+T_j^-$ such that T_i^- is blocking the progress of T_j^- . There is a deadlock iff the graph has a cycle [Holt, KC]. Deadlock prevention schemes such as those in [BG,RS2] can also be used. The system should keep track of the two types of deadlock separately. To resolve deadlocks caused by certify-locks, the system should force one or more transactions to give up enough of their certify-locks to break the deadlock; these transactions can try later to get these locks back. To break deadlocks caused by C1 and C2, the system must abort one or more transactions. (Cascading abort is possible if the algorithm allows transactions to read uncertified versions.)

The algorithm induces the following log properties. These properties form our formal definition of the *mv locking algorithm*. Let L be an mv log over $\{T_0, \ldots, T_n\}$. And let us augment L with symbols that represent important events in the algorithm, specifically: for each T_i , let c_i

represent the event "T_i is declared to be certified"; for each version x_i written by T_i, let $cl_i[x_i]$ represent, "The dbs sets a certify-lock on x_i for T_i"; and for each x_i , let $c_i[x_i]$ represent, " x_i is certified."

 $\underline{\text{L1.1}}$ For every $\underline{\mathbf{T}}_{i}$, $\underline{\mathbf{c}}_{i}$ follows all of $\underline{\mathbf{T}}_{i}$'s read's and writes.

<u>L1.2</u> For every x_i written by T_i , $cl_i[x_i] \le c_i \le c_i[x_i]$.

Property Ll says that a transaction is certified after it executes; all certify-locks must be obtained before the transaction is certified; and the transaction must be certified before its versions are certified.

 $\underline{L2.1}$ Every $cl_{\underline{i}}[x_{\underline{i}}]$ and $cl_{\underline{i}}[x_{\underline{i}}]$ are < related.

L2.2 For every x_i and x_j , if $cl_i[x_i] < cl_i[x_j]$ then $c_i[x_i] < cl_i[x_j]$. L2 says that certify locks conflict—two transactions cannot simultaneously hold certify-locks on the same data item.

 $\underline{L3.1}$ Every $r_k[x_i]$ and $c_i[x_i]$ are < related.

L3.2 For every $r_k[x_j]$ and $w_i[x_i]$, $i \neq j$, if $c_i[x_i] < r_k[x_j]$ and $c_j[x_j] < r_k[x_j]$, then $c_i[x_i] < c_j[x_j]$.

L3 expresses the rule for translating reads. If x_j is already certified at the time $r_k[x_j]$ occurs, then x_j is the *last* certified version at that time.

<u>L4.1</u> For every $r_k[x_j]$, $k \neq j$, $c_j[x_j] < c_k$.

<u>L4.2</u> For every $r_k[x_j]$ and $w_i[x_i]$, $i \neq j$, if $r_k[x_j] < c_i[x_i]$ and $c_j[x_j] < c_i$, then $c_k < c_i$.

These last properties are certification conditions Cl and C2, respectively.

The following lemmas extract useful properties from L1-L4.

LEMMA 1. Let T_i and T_j be transactions that write x. Then $either \ cl_i[x_i] < c_i < c_i[x_i] < cl_j[x_j] < c_j < c_j[x_j],$ $or \ cl_j[x_j] < c_j < c_j[x_j] < cl_i[x_i] < c_i < c_i[x_j].$

Proof. L2.1 requires that $cl_i[x_i]$ and $cl_j[x_j]$ be < related. Suppose $cl_i[x_i] < cl_j[x_j]$. By L1.2, $cl_i[x_i] < c_i < c_i[x_i]$; by L2.2, $c_i[x_i] < cl_j[x_j]$; by L1.2 again, $cl_j[x_j] < c_j < c_j[x_j]$. This establishes the first possibility permitted by the Lemma. If $cl_j[x_j] < cl_i[x_i]$, the same argument establishes the second possibility.

LEMMA 2. Properties L1-L4 imply

L5. For every
$$r_k[x_j]$$
, $k \neq j$, $c_j < c_k$.

L6. For every
$$r_k[x_j]$$
 and $w_i[x_i]$, $i \neq j$, either $c_i < c_j$ or $c_k < c_i$.

<u>Proof.</u> (L5). By L1, $c_j < c_j [x_j]$. By L4.1, $c_j [x_j] < c_k$. L5 follows by transitivity.

(L6) Using logical manipulation we can express L3.2 as

$$\frac{\text{L3.2'}}{\text{c}_{i}[x_{i}]} < r_{k}[x_{j}]) \Rightarrow (c_{i}[x_{i}] < r_{k}[x_{j}]) \land \neg (c_{j}[x_{j}] < r_{k}[x_{j}])$$

$$\lor (c_{i}[x_{i}] < c_{j}[x_{j}]) .$$

By L3.1, the first term on the right hand side simplifies to

$$(c_{i}[x_{j}] < r_{k}[x_{j}]) \wedge (r_{k}[x_{j}] < c_{j}[x_{j}])$$
.

By transitivity, this implies $(c_{i}[x_{i}] < c_{j}[x_{j}])$, and so the entire right hand side implies $c_{i}[x_{i}] < c_{j}[x_{j}]$. By Lemma 1, this implies $c_{i} < c_{j}$. So L3.2' implies

$$\underline{\text{L3.2"}} \quad (c_{\underline{i}}[x_{\underline{i}}] \le r_{\underline{k}}[x_{\underline{j}}]) \rightarrow c_{\underline{i}} \le c_{\underline{j}}.$$

Similarly, we can express L4.2 as

L4.2'
$$(r_k[x_j] < c_i[x_i]) \Rightarrow \neg(c_j[x_j] < c_i) \lor (c_k < c_i)$$
.

By Lemma 1, $c_j[x_j]$ and c_i are < related. So the first term on the right hand side simplifies to $(c_i < c_j[x_j])$. By Lemma 1, again, this is equivalent to $c_i < c_j$. So L4.2' is equivalent to

$$\underline{\text{L4.2"}} \quad (r_{k}[x_{j}] < c_{i}[x_{i}]) \Rightarrow c_{i} < c_{j} \lor c_{k} < c_{i} .$$

L3.1 requires that $r_k[x_j]$ and $c_i[x_j]$ be < related. This lets us drop the left hand sides of L3.2" and L4.2", combining them into

For every $r_k[x_j]$ and $c_i[x_i]$, $c_i < c_j \lor c_k < c_i$. Since $c_i[x_i]$ exists iff $w_i[x_i]$ exists, L6 follows.

We now prove that any log satisfying these properties is 1-SR. In other words, my locking is a correct concurrency control algorithm.

MULTIVERSION LOCKING THEOREM. All logs produced by the mv locking algorithm are 1-SR.

<u>Proof.</u> Let L be a log produced by the algorithm. Define a version order by: $x_i << x_j$ implies $c_i < c_j$. We prove that all edges in MVSG(L,<<) are in certification order: if $T_i \to T_j$ is an edge, then $c_i < c_j$.

Let $T_i \to T_j$ be an edge of SG(L). This edge corresponds to a reads-from situation, i.e., for some x, T_j reads-x-from T_i . By L5, $c_i < c_j$.

Consider any edge introduced by rule (*) of the MVSG definition. Let $\mathbf{w_i}[\mathbf{x_i}]$, $\mathbf{w_j}[\mathbf{x_j}]$, and $\mathbf{r_k}[\mathbf{x_j}]$ be the operations stipulated by rule (*). There are two cases. (1) $\mathbf{x_i} << \mathbf{x_j}$. Then the edge is $\mathbf{T_i} + \mathbf{T_j}$. $\mathbf{c_i} < \mathbf{c_j}$ comes from our definition of <<. (2) $\mathbf{x_j} << \mathbf{x_i}$. Then the edge is $\mathbf{T_k} + \mathbf{T_i}$. By L6, either $\mathbf{c_i} < \mathbf{c_j}$ or $\mathbf{c_k} < \mathbf{c_i}$. The first option is impossible, since the definition of << requires $\mathbf{c_j} < \mathbf{c_i}$. So, $\mathbf{c_k} < \mathbf{c_i}$ as desired.

This proves that all edges in MVSG(L, <<) are in certification order. Since the certification order is embedded in a partial order (namely L), it follows that MVSG(L, <<) is acyclic. So, by the 1-Serializability Theorem, L is 1-SR.

The Stearns and Rosenkrantz algorithm [SR] differs from ours in two respects. They allow at most one uncertified version of a data item to exist at any point in time, by requiring that Write operations set write-locks. Consequently, their algorithm never needs more than two versions of any data item: one certified version and at most one uncertified version. This fits nicely with database recovery [Gray]. Stearns and Rosenkrantz identify the certified version of a data item with its "before-value", and the uncertified version with its "after-value." The other difference involves deadlock handling. Their sporithm uses an interesting new deadlock avoidance scheme based on timestamps.

The Bayer et al. algorithm [BEHR, BHR] also uses at most two versions of each data item. As in [SR], the versions of a data item are identified with its before- and after-values. Unlike [SR], they use the blocking graph to help translate data item reads into version reads. They prove that they can always select a correct version to read. That is, reads never cause a log to become non-1-SR and never cause deadlocks. This is a good property since it allows read-only transactions (queries) to run with little synchronization delay and no danger of deadlock.

MULTIVERSION MIXED METHOD

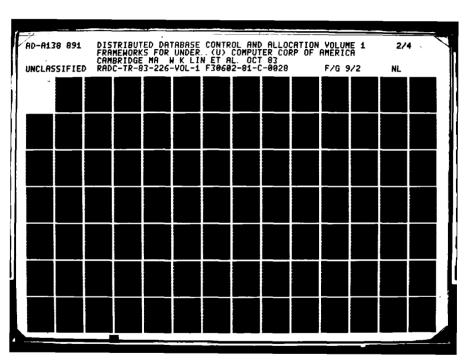
Prime Computer, Inc. has developed an interesting multiversion algorithm [Dubo]. Prime's algorithm, like those at the end of Section 5, integrates concurrency control with database recovery. Unlike those algorithms, Prime's algorithm can exploit multiple certified versions of data items. Computer Corporation of America has adopted Prime's algorithm for its Adaplex dbs [CFLNR]. This section studies a generalization of Prime's algorithm.

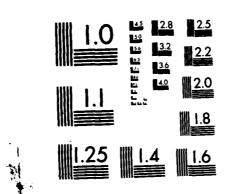
The algorithm we study is called a mixed method. A mixed method is a concurrency control algorithm that combines locking with timestamping [BG]. Mixed methods incroduce a new problem: consistent timestamp generation. A timestamping algorithm uses timestamps to order conflicting transactions; intuitively, if T_i and T_j conflict, then T_i is synchronized before T_j iff TS(i) < TS(j). A locking algorithm orders transactions on-the-fly; intuitively, if T_i and T_j conflict, then T_i is synchronized before T_j iff $C_i < C_j$. To combine locking and timestamping, we must render their synchronization orders consistent.

Our algorithm uses mv timestamping to process read-only transactions (queries). The algorithm uses mv locking to process general transactions (updaters). Queries and updaters are assigned timestamps satisfying two properties:

- 1. Let T_i and T_j be updaters. If $c_i < c_j$ then TS(i) < TS(j).
- 2. Let T_q be a query and T_i an updater. If $r_q[x_k] < w_i[x_i]$ then TS(q) < TS(i).

A consistent timestamp generator is any means of assigning timestamps that satisfy these properties.





MICROCOPY RESOLUTION TEST CHART NATIONAL BUREAU OF STANDARDS-1963-A

Our algorithm uses a Lamport clock to generate consistent timestamps.

Recall the discussion of distributed systems from Section 2. A

Lamport clock assigns a number to each event (called its time) subject to two conditions.

- LCl. If e and f are events of the same process and e happened
 before f, then time(e) < time(f).</pre>
- LC2. If e is the event "Process P sends message M" and f is
 the event 'Process Q receives M", then time(e) < time(f).</pre>

LC1 is easily achieved using clocks or counters local to each process. LC2 can be implemented by stamping each message with the local clock time when it was sent; if a process Q receives a message whose time t is greater than Q's local time, Q pushes its clock ahead to t.

LCl and LC2 imply

LC. Let e and f be events in a distributed system. If e < f then time(e) < time(f) [Lamp].</p>

LC is precisely the condition we need to generate consistent timestamps. When an updater T_i is certified, the process that certifies it assigns $TS(i) = time(c_i)$. By LC, $c_i < c_j$ implies $time(c_i) < time(c_j)$; hence TS(i) < TS(j) as desired. When a query T_q begins executing, we assign $TS(q) \le the$ current Lamport time. So for all reads $T_q[x_k]$, $TS(q) < time(T_q[x_k])$. Consider any write $W_i[x_i]$ such that $T_q[x_k] < W_i[x_i]$. By locking property L1 (see Section 5), $W_i[x_i] < c_i$, so by transitivity $T_q[x_k] < c_i$. By LC this implies $time(T_q[x_k]) < time(c_i)$; hence TS(q) < TS(i) as desired.

We now describe the algorithm in detail.

^{*}The system maintains a Lamport clock.

^{*}Updaters use the mv locking algorithm of Section 5.

- When an updater T_i is certified, the system assigns $TS(i) = time(c_i)$. This timestamp is transmitted to all versions that T_i wrote. Thus, certified versions have timestamps, but uncertified versions don't.
- When a query T_q begins executing, the system assigns $TS(q) \le the$ current time.
- Consider any read by T_q , $r_q[x]$. As in Section 4, we want to translate this into $r_q[x_k]$ where x_k is the version of x with largest timestamp < TS(q). But, some care is needed since uncertified versions don't have timestamps. Let t be a lower bound on the possible timestamps of any uncertified x versions. For instance, let $t = \min\{time(cl_i[x_i]) | x_i$ is uncertified). Since $cl_i[x_i] < c_i$, time $(cl_i[x_i])$ is a lower bound on time $(c_i) = TS(i)$; therefore t is a lower bound on the timestamps of any uncertified x_i .
- Consider $r_q[x]$ again. If x has no uncertified versions, or if $TS(q) < t, \text{ then } r_q[x] \text{ reads the version } x_k \text{ of } x \text{ with largest timestamp} < TS(q); else <math>r_q[x]$ waits until the condition is satisfied. (This will eventually happen.)

The log properties induced by the algorithm are a simple combination of the properties induced by mv timestamping and locking. The correctness proof is similar to those in Sections 4 and 5.

MULTIVERSION MIXED METHOD THEOREM. All logs produced by the mv mixed method are 1-SP.

Prime's algorithm differs from ours in two respects. Most importantly, Prime's algorithm doesn't use explicit timestamps. All certify events are < related; i.e. c_0, \ldots, c_n are totally ordered. The algorithm maintains

a list, CL, of all transactions that have been certified; when T_i is certified, its identifier, i, is included in CL. When a query T_q begins executing, it makes a copy of CL, denoted CL(q). When T_q issues a read, $T_q[x]$, it reads $T_q[x]$, it reads $T_q[x]$ where $T_q[x]$ is the latest version (w.r.t.<) of $T_q[x]$ of that $T_q[x]$ we can analyze this behavior as a special case of our mixed method. Imagine that each updater $T_q[x]$ is assigned a timestamp equal to its place in the certification total order; i.e. $T_q[x]$ is the t-th transaction to be certified. Imagine that $T_q[x]$ is assigned the timestamp $T_q[x]$ is assigned the timestamp $T_q[x]$ is a consistent way of assigning timestamps. If we now run $T_q[x]$ under our algorithm, it reads the same versions as under Prime's algorithm. Since our algorithm is 1-SR, so is Prime's.

The other difference is that Prime uses a restricted form of multiversion locking for updaters, namely two phase locking [EGLT]. Write operations set write-locks, so no data item ever has more than one uncertified version. And, once T_i writes x, no updater T_j reads x until T_i is certified, and vice versa. Consequently, every updater can be certified as soon as it finishes executing.

The net effect is that queries and updaters are totally decoupled. Queries never delay or cause the abort of updaters, and updaters never delay or cause the abort of queries.

Prime's algorithm is most naturally implemented in a centralized dbs because of the need to totally order certify events.

The following variant is more suitable for a distributed dbs

*The system maintains a Lamport clock

*Updaters use two phase locking, hence can be certified as soon as each finishes executing. The system assigns $TS(i) = time(c_i)$, as in the general algorithms.

*Queries are processed using timestamps, exactly as in the general algorithm.

This algorithm decouples and updaters almost as fully as Prime's algorithm. Queries never delay or abort updaters, and updaters never abort queries. But an updater can delay a query under one condition: if a query T_q reads x, updater T_i has a certify-lock on x, and TS(q) the time of that certify-lock, then T_q must wait until T_i certifies x.

CONTRACT GOODS OF THE STATE OF

7. CONCLUSION

This paper has studied the concurrency control problem for multiversion databases. Multiversion databases add a new aspect to concurrency control.

Transactions issue operations that specify data items (e.g., read(x), write(x)); the system must translate these into operations that specify versions. In a single version database, concurrency control correctness depends on the order in which read's and write's are processed. In a multiversion database, correctness depends on translation as well as order.

We have extended concurrency control theory to account for the translation aspect of multiversion databases. The main idea is one-copy serializability: an execution of transactions in a multiversion database is one-copy serializable (1-SF) if it is equivalent to a serial execution of the same transactions in a single version database. A multiversion concurrency control algorithm is correct if all of its executions are 1-SR. We derived effective necessary and sufficient conditions for an execution to be 1-SR; these condition use the concept of version order. We gave a graph structure, multiversion serialization graphs (MVSG's), that helps check these conditions. Once a version order is fixed, an execution is 1-SR iffits MVSG is acyclic. MVSG's are analogous to the serialization graphs widely used in single version concurrency control theory.

We applied the theory to three multiversion concurrency control algorithms. One algorithm use timestamps, one uses locking, and one combines locking with timestamps. The timestamping algorithm is due to Reed [Reed]. The locking algorithm was inspired by (and generalizes) the work of Bayer et al. [BEHR, BHR] and Stearns and Rosenkrantz [SR]. The combination algorithm generalizes an algorithm developed by Prime Computer, Inc. [Dubo] and used by Computer Corporation of America [CFLNR].

REFERENCES

- [BEHR] Bayer, R., E. Elhardt, H. Heller, and A. Reiser, "Distributed Concurrency Control in Database Systems," Proc. Sixth Int'l Conf. On Very Large Data Bases, IEEE, N.Y., 1980, pp. 275-284.
- [BHR] Bayer, R., H. Heller, and A. Reiser, "Parallelism and Recovery in Database Systems," ACM Trans. on Database Syst. 5, 2 (June 1980), pp. 139-156.
- [BG] Bernstein, P.A. and N. Goodman, "Concurrency Control in Distributed Database Systems," ACM Computing Surveys 13, 2 (June 1981), pp. 185-221.
- [BSW] Bernstein, P.A., D.W. Shipman, and W.S. Wong, "Formal Aspects of Serializability in Database Concurrency Control," *IEEE Trans. on Software Eng.* SE-5, 3 (May 1979), pp. 203-215.

TRICKER DESCRIPTION OF THE PROPERTY ASSESSED FOR SECURIAL FOR THE PROPERTY ASSESSED FOR THE PROPERTY ASSESSED FOR

- [Casa] Casanova, M.A. The Concurrency Control Problem of Database Systems, Lecture Notes in Computer Science, Vol. 116, Springer-Verlag, 1981 (originally published as TR-17-79, Center for Research in Computing Technology, Harvard University, 1979).
- [CFLNR] Chan, A., S. Fox, W.T. Lin, A. Nori, and D. Ries, "The Implementation of an Integrated Concurrency Control and Recovery Scheme," Proc. 1982 ACM SIGMOD Conf., ACM, N.Y.
- [Dubo] Dubourdieu, D.J., "Implementation of Distributed Transactions," Proc. 1982 Berkeley Workshop on Distributed Data Management and Computer Networks, pp. 81-94.
- [EGLT] Eswaran, K.P., Gray, J.N., Lorie, R.A., and Traiger, I.L. "The Notions of Consistency and Predicate Locks in a Database Systems," Communications of the ACM, Vol. 19, No. 11, November 1976.
- [Gray] Gray, J.N. "Notes on Database Operating Systems," Operating Systems:

 An Advanced Course, Vol. 60, Lecture Notes in Computer Science,

 Springer-Verlag, N.Y., 1978, pp. 393-481.
- [GJ] Garey, M.R. and D.S. Johnson, Computers and Intractability: A Guide to the Theory of NP-Completeness, W.H. Freeman and Company, San Francisco, 1979.
- [Holt] Holt, R.C., "Some Deadlock Properties of Computer Systems," Computing Surveys 4, 3 (Dex. 1972), pp. 179-195.
- [KC] King, P.F., and Collmeyer, A.J. "Database Sharing--An Efficient Mechanism for Supporting Concurrent Processes," *Proc.* 1974 NCC, AFIPS Press, Montvale, New Jersey, 1974.

- [Lamp] Lamport, L., "Time, Clocks, and the Ordering of Events in a Distributed System," Comm. of the ACM 21, 7 (July 1978), pp. 558-565.
- [Papa] Papadimitriou, C.H., "Serializability of Concurrent Updates," Journal of the ACM 26, 4 (Oct. 1979), pp. 631-653.
- [PBR] Papadimitriou, C.H., Bernstein, P.A., and Rothnie, J.B., Jr. "Some Computational Problems Related to Database Concurrency Control," Proc. Conf. on Theoretical Computer Science, Waterloo, Ontario, August, 1977.
- [PK] Papadimitriou, C.H., and P.C. Kanellakis, "On Concurrency Control by Multiple Versions," *Proc. 1982 ACM SIGACT-SIGMOD Symp. on Principles of Database Syst.*, March 1982.
- [Reed] Reed, D., Naming and Synchronization in a Decentralized Computer System, Technical Report MIT/LCS/TR-205, M.I.T., Dept. of Electrical Engineering and Computer Science, Sept. 1978.
- [RSL] Rosenkrantz, D.J., R.E. Stearns, and P.M. Lewis II, "System Level Concurrency Control for Distributed Database Systems," ACM Trans. cr. Database Syst., 3, 2 (June 1978), pp. 178-198.
- [Silb] Silberschatz, A., "A Multiversion Concurrency Control Scheme with no Rollbacks," Proc. ACM Symmosium on Principles of Distributed Computing, Aug. 1982.
- [SLR] Stearns, R.E., P.M. Lewis II, and D.J. Rosenkrantz, "Concurrency Controls for Database Systems," *Proc. 17th Symp. on Foundations of Computer Science*, IEEE, N.Y., 1976, pp. 19-32.
- [SR] Stearns, R.E., and D.J. Rosenkrantz, "Distributed Database Concurrency Controls Using Before-Values," *Proc.* 1981 ACM-SIGMOD Conf., ACM, N.Y., pp. 74-83.

SECTION IV

PERFORMANCE ANALYSIS OF CONCURRENCY
CONTROL METHODS IN DATABASE SYSTEMS*

Annie W. Shum

Paul G. Spirakis

^{*}Published in Performance '81, F.J. Klystra (editor), North Holland Publishing Co., 1981.

Abstract

Although concurrency control in both centralized and distributed systems is a widely studied problem, not many formal quantitative methods are known for analyzing and comparing the performance of the proposed algorithms which are usually complex, not described in standard terminology and each of them has different assumptions regarding the underlying environment. In this paper we analyze dynamic 2PL, proceeding from a simple deadlock prevention 2PL (which gives worst case bounds on the performance of any deadlock preventing 2PL) to the general case of two phase locking where deadlocks are allowed. The 2PL methods considered here are dynamic.

1.0 INTRODUCTION

Although concurrency control in both centralized and distributed systems is a widely studied problem, not many formal quantitative methods are known for analyzing and comparing the performance of the proposed algorithms which are usually complex, not described in standard terminology and each of them has different assumptions regarding the underlying environment (inputs, communication rules, etc.). Fortunately, it is possible to decompose every concurrency control problem into two major subproblems, namely read-write and write-write synchronization [B-G, 80]. Every algorithm must include a subalgorithm for solving each of these problems. Current research indicates that only a few subalgorithms are possible. In fact, most of them are variations of two basic techniques: two phase locking (2PL) and timestamp ordering (T/O). Therefore, it is important to develop quantitative methods for analyzing the performance of these two techniques.

In this paper we analyze dynamic 2PL, proceeding from a simple deadlock prevention 2PL (which gives worst case bounds on the performance of any deadlock preventing 2PL) to the general case of two phase locking where deadlocks are allowed. The 2PL methods considered here are dynamic. Locks are requested in a distributed manner over the lifetime of transaction, as found in most 2PL methods. The analysis of simple 2PL is used primarily to serve as worst case performance bound for any 2PL methods. In the analysis of the general 2PL, results indicate that the rate of deadlocks at steady state is approximately proportional to the mean number of transactions in the system. Our techniques for analyzing 2PL apply to both centralized and distributed systems by suitably adjusting certain assumptions, and parameters.

2.0 ESSENTIAL CONCEPTS AND DEFINITIONS

A database consists of a collection of logical data items x, y, z,... etc. A logical database state is an assignment of values to logical data items. Users interact with the database by executing transactions. Each transaction is considered to be a sequence of Read and/or Write operations. The readset (set of logical data items that the transaction reads) and the writeset (set of logical data items that the transaction writes) are part of the information content of the transaction. In case of distributed systems, the "stored" readset and writeset refer to the stored copies of the logical data items in the various places. Two transactions conflict if the (stored) readset or writeset of one intersects the (stored) writeset of the other (read-write and write-write conflicts). Two transactions need synchronization only if they conflict [B-G, 80].

Synchronization in update processing is necessary to maintain the consistency of databases. Different synchronization techniques can be applied to synchronize read-write and write-write conflicting transactions, as soon as they induce the same total order in the transactions. Two phase locking and timestamp ordering are the two most common methods to solve these subproblems.

2PL synchronizes read and write operations by explicitly detecting and preventing conflicts. The essentials of the method can be outlined as follows: Before reading or writing on a data item x, a transaction must own a lock on x.

There are two rules for ownership of locks:

- (1) Different transactions cannot simultaneously own locks that conflict (they conflict if they are both on the same item).
- (2) Once a transaction surrenders ownership of a lock, it may not obtain additional locks. The second rule forces transactions to obtain locks in a two-phase manner (a growing and a shrinking phase in terms of the set of locks owned by the transaction). 2PL is a correct synchronization technique in the sense that logs which are two phase locked, are serializable [BSW, 79].

2PL may block a transaction by causing it to wait for an unavailable lock. In case of distributed systems, a transaction may have to wait in a site, even if the site is currently idle. If this blocking is uncontrolled, deadlocks may occur.

A waits-for graph is a directed graph indicating which transactions are waiting for which other transactions. Its nodes represent transactions and its edges the wait-for relationship. (An edge is drawn from transaction T_i to T_j (i \neq j) iff T_i is waiting for a lock currently owned by T_j). A deadlock in the database system corresponds to a cycle in this graph. If a deadlock exists, one (or more) transactions in the cycle are restarted. Avoidance of cyclic restarts is possible if, for example, the "youngest" transaction in the cycle is always restarted.

There are two mechanisms to resolve deadlocks, namely deadlock prevention and detection.

^{1.} The execution of transactions is modelled by a set of logs, each of which indicates the order in which reads and writes are processed at one data module.

3.0 DEADLOCK PREVENTION AND DETECTION TECHNIQUES IN 2PL ALGORITHMS

Deadlock prevention is a "cautious" scheme, since it restarts a transaction every time the system suspects that a deadlock will occur. Suppose Transaction T_1 is blocked by T_j . A procedure test (T_i,T_j) is generally applied to the blocking pair. If the pair passes the test, then T_i is permitted to wait for T_j . Else, one or the other is restarted. If $T_i(T_j)$ is restarted, we have a non-preemptive (preemptive) deadlock prevention method. The test (T_i,T_j) must ensure that the addition of the edge (T_i,T_j) to the waits-for graph cannot introduce a cycle.

Many tests have this property. The simplest example is when T_i is always restarted (i.e., the blocking pair never passes the test). This "simple" 2PI causes no waiting but many restarts. In terms of restarts it may be consite to be a worst case of the prevention techniques. The analysis of the perimance of simple 2PL is outlined in Section 5.0.

Concurrency control algorithms which induce fewer restarts are priority as a estamp based [RLS, 78], [Th, 79] (wait-die, wound-wait 2PL).

In deadlock detection schemes, transactions are allowed to wait for each other in an uncontrolled manner and only abort transactions when a deadlock actually occurs. The real-time waits-for graph is searched for cycles and (at least) one transaction in each cycle is aborted. In Section 6.0 we analyze this general 2PL by describing the evolution of the waits-for graph as a Markovian process. The steady-state deadlock rate is found to be approximately proportional to the mean number of transactions in the system and the steady state conflict rate approximately proportionately to the mean of the product of the number of transactions in the system and the number of unblocked transactions. Prior to the discussion of the models, let us begin with a brief review of the current research in this area.

4.0 PAST WORK

Garcia Molina [GM, 79] compared variants of centralized 2PL to the majority consensus algorithm of [Th, 78], mostly based on simulation studies. He concluded that centralized ZPL out-performs the Thomas's algorithm under all tested conditions. His assumptions included predeclaration of readsets and writesets and a fully redundant database. The analysis was limited to optimistic situations in which run-time conflict never occurred. Ries [R, 79] analyzed by simulation various forms of centralized 2PL and variations of distributed 2PL. For all variations, static locking was assumed and the simulations were set up in a way that no deadlocks occurred and run time conflicts were very rare. Consequently, the main result was that all methods had similar behavior. One of the early analytic approaches to performance evaluation of concurrency control schemes was by Gelenbe and Sevcik [Ge-S 78]. Their analysis was mainly on timestamp ordering methods. However, their assumptions about the update rules are different from the current implementations. They defined measures of "coherence" (how much the values of various copies of the data agree) and "promptness" (how up-to-date the data is). Their results were based on assumptions about transmission delays and state that coherence improves rather quickly and promptness drops slowly under all methods considered. Menasce and Nakanishi [M-N, 79] examined "conflict oriented" methods analytically and locking by doing simulations. The analysis was mostly done on "certification methods." (A transaction has three phases: read phase, a computation phase, and a test and update phase. It can be restarted only at the end of the last phase. After the transaction has visited all the places it wants, its timestamp is compared against timestamps obtained during the read phase). Unfortunately, their assumptions were not very clear and the model analysis seemed to be oversimplified. D. Potier and P. Leblanc [D-1, 80] analyzed the case of centralized 2PL with static looking (a transaction is allowed to enter the system only if all its requested locks are available). The probability that a new transaction is granted its locks, given k other transactions already active, was found. A general limitation of all previous analysis on 2PL is that only static looking is examined. Although this simplifies the analysis with respect to time dependence of state changes of the system, the issue of deadlocks and restarts will be avoided under static locking. In order to be able to quantify the degradation on performance due to restarts or deadlocks, it is important to study dynamic locking. In addition, since most real systems use dynamic locking, the primary objective of our analysis is to focus on the performance of dynamic concurrency control systems including the discussion of the rate of deadlocks and restarts.

5.0 ANALYSIS OF THE "SIMPLE" ZPL AND WORST CASE BOUNDS

5.1 The Model

The physical database is assumed to be composed by M granules (data items) with no redundancy. In the case of a centralized multiprogramming system the granules are located in some storage device (e.g., disk). In case of a distributed database, there is one granule per site and the M sites together with their communication links form a fully connected graph. Links and processors are assumed completely reliable.

The transactions are characterized by the "hopping behavior." That is, each transaction moves from granule to granule and requests the lock on arrival. If the lock is granted then the transaction "hops" to another place, after spending some time in that granule (site). If the lock is denied then the transaction simply restarts; hence, this is the worst cast of 2PL in terms of number of restarts.

Each transaction selects its move to the next granule at random (independently of other transactions) with probability $(1-\theta)/M$ for each of the M granules² or it completes (exits the system) with probability θ . (Note that this includes revisiting the same granule. When this happens, the transaction does not conflict with itself and simply spends some time in the granule and then it selects the next granule to move by the same process.) On each visit to the granules, the transactions obtain service for a random time interval following the exponential distribution, with service rate μ .

Note that, because of our assumptions, the total lifetime of a transaction (conditioned on no restarts) again follows the exponential distribution of mean

$$\frac{1}{u} = \frac{1}{\theta u^*}$$

(geometric sum of exponentials of same mean).

In case of a restart the transaction releases all its locks instantaneously and follows the same "hopping" behavior. Similarly on total completion the transaction releases its locks instantaneously. The input process (arrival time distribution of transactions) is assumed to be Poisson of rate λ .

^{2 [}L-N, 81] conducted simulations in a non-uniform access model in which 20% of the database are accessed 80% of the time they found that the behavior is very similar with the behavior of the random access model of heavier load. This indicates that the access pattern is not the dominant factor of the overall performance in large databases.

5.2 Justification of the Assumptions

The Poisson arrival process has been widely used to model open transaction streams. In a distributed database, this assumption is further justified because the input process is the sum of many point processes (one for each site) and this sum converges to a Poisson process under weak conditions. Elimination of redundancy is analogous to the primary copy policies of locking in redundant systems. The "hopping" transaction model implies that at any time instant t, a transaction can be active only at one granule and that it requests locks sequentially. This is true in most centralized databases. Due to lack of commercial distributed DBMS, this assumption is yet to be tested in cases of distributed systems. Having described the overall model, we shall now proceed with the analysis study.

Previously, the "hopping" transaction model has been assumed in [R-S-L, 78] for distributed database systems. It is a necessary assumption for the conflict driven restart methods, including wait-die and wound-wait, to work in distributed systems. (See [R-S-L,78] and [B-G,78]). The authors justify the use of the "hopping" model by considering systems in which program are executed at single sites but can call other programs as subroutines at other sites. The execution of a program together with all its associated subroutine calls are treated as the execution of a single process. At any time, the "active" site is the site of the subroutine being executed and the inactive sites are those where programs are waiting for subroutines to return or where subroutines have finished their execution. When a program at one site calls a program at another site, both programs together are responsible for maintaining global consistency.

There are other reasons for the "hopping" assumptions, one of which is the difficulty for the impelementation of a different transaction behavior in distributed systems: The users would have to write "parallel" programs, producing "transaction incarnations" active at different sites at the same time. This would require sophisticated synchronization methods for "chasing" the incarnations in case of restarts, deadlock avoidance etc. (For such systems there is a potential danger that a process will be rolled back at one site and made permanent at another site, thereby violating consistency requirements.

Furthermore it is necessary for the users to have a good knowledge of information contained in sites and of communication delay parameters of the system. Consequently, no such systems are currently implemented.

Having described the overall model, we shall now proceed with the analysis study.

5.3 Performance Analysis of the "Simple" 2PL

In the performance study we shall consider the following: (1) Nontrivial lower bounds for the steady state probability that a transaction can complete without any restarts; (2) An upper bound on the overage number of restarts per transaction; and (3) mean response time of a transaction (restarts included).

5.3.1 Analysis of the System

Assume that the system has a steady state and let p be the steady state probability that a transaction will complete without any restarts. Consider at time t a "marked" transaction T_1 obtaining service at a granule. T_1 is said to be active at t. Let $Y_k(t,dt)$ be the $Prob\{T_1$ will conflict at (t,t+dt) given that T_1 is active at t and there are k transactions present in the system}. Then, $Prob\{T_1$ will conflict at $(t,t+dt)\}$ is simply the unconditional probability of $Y_k(t,dt)$ which we denote by Y(t,dt). Assume that Y(t,dt) has a steady state value Y-dt. From that assumption,

$$\gamma \cdot dt = \lim_{t \to \infty} \frac{\infty}{x} \gamma_k(t, dt) \cdot p(k, t)$$

where $p(k,t) \stackrel{\text{def}}{=} Prob\{k \text{ transactions in the system at time } t\}$. Consequently, the average rate of conflicts at steady state is a constant (namely γ) and hence the probability f(T) of no conflicts (and hence no restarts) in the lifetime T of a transaction is (by basic properties of the Poisson process),

$$f(T) = e^{-\gamma \cdot T}$$

In addition,

Prob{T₁ does not restart during its whole life}

• (f(T) Prob{T < T < T +dT}dT

$$= \int_{\tau=0}^{\infty} f(\tau) \cdot Prob\{\tau \leq T \leq \tau + d\tau\}d\tau$$

$$= \int_{\tau=0}^{\infty} e^{-\gamma \tau} \cdot \mu e^{-\mu \tau} d\tau = \frac{\mu}{\mu + \gamma}$$

So, we conclude that

$$\underline{p} = \frac{\mu}{\mu_{+}\gamma} \tag{2}$$

To find an expression for the restart rate at steady state, we begin by pointing out that in order for T_1 to conflict at (t,t+dt) (and consequently restart) it is necessary for the following events to occur: (1) T_1 completes service at its current granule within (t,t+dt) without departing from the system, and (2) The next granule T_1 chooses must be one of the granules already locked by one of the other k-1 transactions in the system. Let $T_2,T_3,\ldots,T_k,k\geq 2$ be the other transactions and let L_2,\ldots,L_k be the number of distinct granules locked at t by T_2,\ldots,T_k respectively. Then

$$\gamma_k(t,dt) = (\mu'dt)(1-\theta) \frac{L_2+\cdots+L_k}{M}$$

Let us approximate the sum $L_2+\cdots+L_k$ by the sum of the mean number of granules visited by each transaction (given no restarts). Since this mean is bounded above by $1/\theta$, we shall approximate $L_2+\cdots+L_k$ by $(k-1)/\theta$. This approximation is pessimistic since $1/\theta$ is an upper bound achieved only in systems with infinite number of granules—and does not refer to distinct granules. Another reason for this approximation to be pessimistic arises from the fact that $(k-1)\cdot 1/\theta$ increases with k for every value of k while $L_2+\cdots+L_k$ is bounded by M-1. Fortunately, it is desirable to be pessimistic here since the results of the "simple" 2PL will be worst case performance bounds for any 2PL method. Using this approximation, we can obtain the following:

$$\lim_{t\to\infty} \sum_{k=1}^{\infty} \gamma_k(t,dt) \cdot p(k,t) = \lim_{t\to\infty} \sum_{k=1}^{\infty} a(k-1)dt \cdot p(k,t)$$
where $a = \frac{\mu^*(1-\theta)}{\theta M} = \frac{\mu(1-\theta)}{\theta^2 M}$

$$\Rightarrow \gamma \cdot dt = a \cdot dt \begin{bmatrix} E(k) - 1 + p(k=0) \\ E(k) - 1 + p(k=0) \end{bmatrix} \cdot dt$$

$$= a \left[E(k) - 1 + p(k=0) \right] \cdot dt$$

where p(k), p(k=0), E(k) are the steady state values of p(k,t), p(0,t) and mean number of transactions in the system at time t.

Substituting Y into 2) we get

Pinally,
$$\underline{p} = \frac{1}{1 + \frac{1 - \theta}{\theta^{2} m} \left(E(k) + p(k=0) - 1 \right)}$$
(3)

Note that Eq. (3) is derived without any assumption about the distribution of transactions in the system. From Eq. (3), we get

$$\underline{P} \ge \frac{1}{1 + \frac{1 - \theta}{\theta^2 m} \cdot E(k)} \qquad (4)$$

Since E(k) is an operational variable, it could be obtained by measurement in most practical systems. So, Eq. (4) can be used as a mean to determine a lower bound on p by measuring E(k). Furthermore, in most real systems the maximum value of E(k) is bounded by a constant (maximum multiprogramming degree due to finite memory considerations, thrashing).

Let C be such a bound on E(k). Then,

$$\frac{\mathbf{p}}{1 + \frac{1 - \theta}{\theta^2 \mathbf{m}}} \quad \mathbf{c} \tag{5}$$

Assuming that the upper bound of E(k) exists and is finite then we can note the following from Eq. (3):

If $\exists m$ constant, $m \ge 1$ such that $\theta \ge m(\sqrt{M})$, then as $M + \infty$ p approaches the constant value

$$\frac{1}{1+\frac{1}{m^2}\cdot E(k)} \tag{6}$$

In other words, if the fraction of the database that each transaction accesses is less than $1/m\sqrt{M}$ then the probability of passing without any restart is nonzero even at high traffic (given our started assumptions).

In the above analysis, one should note that for the case $1/\theta = \Omega(M)$ using $(k-1)/\theta$ as approximation of $L_2+\cdots+L_k$ can be very pessimistic. For example, when $1/\theta = M$, it gives $L_2+\cdots+L_k \simeq M(k-1)$ whereas $L_2+\cdots+L_k \leq M-1$. Thus, in this case it seems to be more appropriate to approximate $L_2+\cdots+L_k$ by its worst case value, namely m-1; then $\Upsilon_k(t,dt)$ becomes $\mu^*\cdot(1-\theta)\cdot(1-1/M)\cdot dt$ (not depending on k). Hence

$$\underline{\mathbf{p}} = \frac{\mu}{\mu + \Upsilon} \geq \frac{\mu'\theta}{\mu'\theta + \mu'(1-\theta)(M-1)/M}$$

$$\Rightarrow \underline{\mathbf{p}} \geq \frac{M\theta}{M\theta + (1-\theta)(M-1)} \tag{7}$$

From Eq. (7), we can note the following:

- (1) $p > \theta > 0$
- (2) The lower bound of Eq. (7) limits to θ as $M \to \infty$
- (3) if $\theta = 1/M$ then $p \ge 1/M-1$

Another useful bound for \underline{p} can be derived from Eq. (3) by using an estimation of $\underline{E}(k)$. Instead of the direct value of $\underline{E}(k)$, by Little's formula,

$$E(k) = \lambda \cdot \overline{T}$$

where

 $_{\mathrm{T}}$ def $_{\mathrm{mean}}$ mean time a transaction spends at the system, and,

$$\bar{T} = \sum_{i=1}^{\infty} \bar{T}_i \cdot p(i-1 \text{ restarts})$$

where

 $T_i \stackrel{\text{def}}{=}$ mean time in system given i-1 restarts.

Since p(0 restarts) = p and

$$\tilde{T}_0 \leq \frac{1}{u'\theta} + u = \frac{1}{u} + u$$

where u is the time to restart a transaction, and $1/\mu$ is the mean time of a transaction in the database given no restarts,

$$\overline{T}_{i} \leq i(1/\mu + u) \quad \forall i$$

Under the assumptions in the analysis, the number of restarts per transaction is geometrically distributed and so

$$p(i-1 restarts) = (1-p)^{i-1} \cdot p$$

Hence,

$$\overline{T} \leq \sum_{i=1}^{\infty} i (1/\mu + u) (1-p)^{i-1} p \leq 1/p(1/\mu + u)$$

$$\mathbf{E}(\mathbf{k}) \leq \frac{\lambda}{\mathbf{p}} (1/\mu + \mathbf{u})$$

and from Eq. (4)

Solving Eq. (8) with respect to p we get

$$\underline{\mathbf{p}} \ge 1 - \frac{1-\theta}{\theta^2 \mathbf{M}} \qquad (1/\mu + \mathbf{u}) \tag{9}$$

If $\exists m$ constant, m > 1, such that $\theta > m (\sqrt{M})^{-1}$ then we get

$$\underline{p} \geq 1 - \frac{1-\theta}{m^2} \lambda (1/\mu + u)$$

and we see that p has nonzero high values even in cases of $\lambda/\mu >> 1$.

In systems where pages are the units of locking, e.g., system R, DMS1100, the value of M can be very large. So let us now consider the cases when M >> 1.

In here, we can assume u to be negligible relative to 1/µ. In this case,

$$E(k) \simeq \frac{\lambda}{\mu p}$$
,

⇒ from Eq. (9)

$$\underline{P} \geq 1 - \frac{1-\theta}{u^2 M} \cdot \frac{\lambda}{\mu} \qquad \text{if} \quad \frac{\lambda}{\mu} < \frac{\theta^2 M}{1-\theta}$$
 (10)

Note that p(k=0) increases as M increases and gets its maximum as $M \to \infty$. In this case, the system can be approximated by an $M[G]^\infty$ system with birth rate λ and death rate $k\mu p$ $(k \ge 2)$, because there is no queuing in the system due to immediate restarts in case of conflicts. Hence the maximum value of p(k=0) is given by $-\lambda/\mu p$ Applying Eq. (3)

$$\underline{p} \geq \frac{1}{1 + \frac{1-\theta}{\theta^2 \mu} \left(\frac{\lambda}{\mu \underline{p}} + e^{-\lambda/\mu \underline{p}} - 1\right)}$$
 (11)

If $1/\theta < \sqrt{M}$ we get from Eq. (11)

$$\underline{p} \geq \frac{1 - \frac{1-\theta}{\theta^2 M} \frac{\lambda}{\mu}}{1 + \frac{1-\theta}{\theta^2 M} (e^{-\lambda/\mu \underline{p}} - 1)}$$

Since $p \le 1$, we get for $\lambda/\mu < 1$

$$\underline{\mathbf{p}} \ge \frac{-\lambda/\mu}{\ln (1-\lambda/\mu)} \tag{12}$$

5.3.2 Overview of the Bounds Obtained for P

From the analysis above, the various bounds obtained for $\,p\,$ hold for certain ranges of values of $\,\lambda/\mu,\,\,\theta\,$ (or both). In cases in which more than one bound equations are applicable, one should select the best lower bound. In order to evaluate the various bounds, we proceed to obtain the exact solution of the model as a Markov process by numerical methods. A description of the method and results follows in Section 5.3.3. We found that the bounds

$$1 - \frac{1-\theta}{\theta^2 M} \cdot \frac{\lambda}{\mu}$$
 for $\frac{\lambda}{\mu} < \frac{\theta^2 M}{1-\theta}$

and
$$\frac{-\lambda/\mu}{\ln(1-\lambda/\mu)}$$
 for $\frac{\lambda}{\mu} < 1$ and $\frac{1}{\theta} \le \sqrt{M}$

approximate the actual value of $\,p\,$ very well from below for $\,M \geq 100$. The bound

$$\frac{M\theta}{M\theta + (1-\theta)(M-1)}$$

yields effective results for small M's (M < 10). If an accurate estimation of E(k) can be produced, then the bound

$$\frac{1}{1+\frac{1-\theta}{\theta^2M}E(k)}$$

approximates p very closely from below in all cases (see Table 3).

Finally, for M < 100, the computational effort for an exact solution is acceptable, and it can be used to get an accurate estimation of p if necessary. Let us now give a brief summary in the following table with the following notation:

$$PL = 1 - \frac{1-\theta}{\theta^2 M} \frac{\lambda}{\mu} , \quad PLOG = \frac{-\lambda/\mu}{\ln(1-\lambda/\mu)}$$

$$PTH = \frac{M}{M + (1-\theta)(M-1)} , \quad PK = \frac{1}{1 + \frac{1-\theta}{\theta^2 M} E(k)}$$

Table 1 M > 100

Table 1 M > 100								
	< √ <u>M</u>	> √M						
₹1	best of PL, PLOG, PTH	best of PL, PTH						
$\leq \frac{\theta^2 M}{1-\theta}$	best of PL, PTH (PLOG if applicable)	best of PL, PTH						
Other high values	best of PTH, PK	best of PK, * PTH						

Only if E(k) can be estimated. (Numerical solution is implicitly assumed for each case).

Table 2 M < 100

M < 10	PTH or exact solution
10 < M < 100	max(PK,PTH) or exact solution

For $\lambda/\mu >> \max(1, \theta^2 M/1-\theta)$ the derivation of analytical tighter lower bounds of **p** is under current investigation.

Evidence from the simulations conducted at CCA [] indicates that performance is considerably degraded fro high values of λ/μ . Consequently, it could be the case that further analysis for such cases may not be necessary for the operational range of existing systems.

5.3.3 The Computational Solution of a Markov Process Describing the Actual System

We developed by numerical methods the exact solution of the global balance state equations for a system consisting of M granules and a CPU (feeding the database) with maximum MPL equal to N. We assume that restarted and new arrived transactions join the queue of the CPU. The system is modelled as a Markov process with state vector (j,k,l) where $j \leq N$ is the number of transactions in the CPU queue and $k \leq M$ is the number of active transactions; l is the total locks held $(k \leq l \leq M)$. We assume that when a transaction completes or restarts, it frees a number of locks which is a random variable uniform in the interval (l, l-k+1). The other parameters of the system include the CPU rate (μCPU) , the input rate λ and the granule service rate μ' , θ is the exit probability, as in the simple 2PL model. The solution of the global balance equations is produced by the power method of finding the eigenvector corresponding to the biggest eigenvalue of a stochastic matrix. This method is preferred because it has minimal storage requirements.

For more details on the method, see [S-78]. Our results show that the lower bounds PLOG, PL work well for $M \ge 100$. PK works well for any value of M, if E(k) is known. For smaller M, either PTH or PK is suggested. For very small M (<50) the computational effort is small and the exact solution can be used as an alternative. All our bounds were validated to be true pessimistic bounds (reference Table 3).

Table 3
Results from the Computational Solution

(1) $M \le 10$, $N \le 10$

μCPU	μ'	λ	N	M	θ	₽	PK
3	2	2	2	2	.2	.78	.089
4	2	5	2	2	.1	.62	.019
3	2	2	3	3	. 2	.60	.1
4	2	5	3	3	.1	. 34	.022
3	2	2	4	3	. 2	. 59	.104
4	2	5	4	3	.1	. 34	.022
3	2	1.9	4	3	.5	.901	.609
3	2	2	2	4	.2	.52	.11
3	2	5	2	4	.1	.24	.2
3	2	1.9	2	4	.5	.9	.66
3	2	2	5	5	.5	.91	.71
3	2	1.9	5	5	. 2	.56	.136
4	2	5	4	5	.1	.28	.26
3	2	1.9	4	5	.5	.91	.711
4	2	5	3	6	.1	. 32	. 32
3	2	1.9	3	6	.5	.92	.74
4	2	5	10	3	.1	.29	.2
3	2	1.1	10	3	.5	.914	.625
3	2	2	2	10	.2	.75	.26
4	2	5	2	10	.1	.52	.17

Note: p = actual value

A TERRESE TERRESE CONSTRUCTION CONTRACTOR PROPERTY OF THE PROP

$$PK = \frac{1}{1 + \frac{1-\theta}{\theta^2 M} E(k)}$$

(2) M > 10, N > 10

μCPU	μ'	λ	N	М	6	<u> </u>	PK	PB
3	2	2.	10	10	. 2	.69	.61	.62
4	2	5	10	10	.1	.49	.37	.15
3	2	1.9	10	10	.5	.88	.72	.62
2	2	.8	30	30	.5	.96	.94	.49
2	2	1.9	30	30	.5	.92	.9	.87
3	2	2	30	30	. 2	.56	.27	. 25
2	2	.8	50	50	.5	.96	.94	.96
2	2	1.1	50	50	.2	.66	.64	. 56
2	2	.8	50	100	.19	.83	.83	.82
2	5	.25	50	100	.05	.07	.07	.05

Note: PB is best of PL, PLOG, PTH.

5.3.4 Upper Bound on the Average Number of Restarts Per Transaction

Let n denote the average number of restarts per transaction. Then

$$\overline{n} = \begin{pmatrix} \infty \\ \Sigma & i & (1-p)^{\frac{i-1}{2}} \end{pmatrix} - 1$$

$$\overline{n} = \frac{1}{p} - 1$$
(13)

From Eq. (3)

$$\overline{n} = \frac{1-\theta}{\theta^2 M} [E(k) + p(k=0) - 1]$$
(14)

From Eq. (4)

$$\frac{1}{n} \leq \frac{1-\theta}{\theta^2 M} \cdot E(k)$$

In general, if PBEST is the best lower bound on p (using Table 1) for each case, then

$$\frac{1}{n} \leq \frac{1}{PBEST} - 1$$

A general conclusion is that if $\exists m$ constant , $m \ge 1$, such that $\cdot \theta > m(\sqrt{M})^{-1}$ then \overline{n} is small for $M \to \infty$. This conclusion applies well to cases where M > 100.

5.3.5 Average Response Time of a Transaction

The average response time \overline{R} of a transaction is bounded above by

$$\overline{n} \cdot (1/\mu + u) = (1/p) (1/\mu + u)$$

For M > 100, we can use

$$\bar{R}$$
 < (1/p) 1/ $\theta\mu$ *

where , as an estimation of p, the best value of PL,PLOG,PTH,PK can be used.

This completes our analysis of simple 2PL, an algorithm which should be viewed as a useful theoretic tool to get worst case performance bounds of any dynamic 2PL method in terms of restarts. In the next section, we shall proceed to the more general dynamic '2PL analysis.

6.0 THE GENERAL CASE OF TWO-PHASE LOCKING AND THE PROBABILITY OF DEADLOCK IN THE STEADY STATE

From the "simple" 2PL analysis, we now consider the more general case of 2PL. We shall begin our discussion of the general case of 2PL with an overview of the mechanism and the related concepts. Primarily we are interested in an analytic model to estimate the steady-state rates of conflicts and deadlock in a database system with general 2PL policy. We assume that each transaction T sequentially requests locks on a randomly selected set of data items. All locks are assumed to be exclusive locks; i.e., T is granted a lock on data item X only if no other transaction currently owns a lock on X. If the lock request is denied, T is placed on a FIFO queue for X. All locks held by a transaction are released simultaneously when the transaction completes. The input process of transactions is assumed Poisson of rate λ . Let us consider the waits-for graph at time t. The nodes in this graff represent all active transactions and the edges indicate which transactions are waiting to obtain locks. In particular, there is an edge from T to T' iff

- (1) T' is immediately in front of T on the queue for some data item X or
- (2) T is the first transaction on the queue for X and T' currently owns a lock on X.

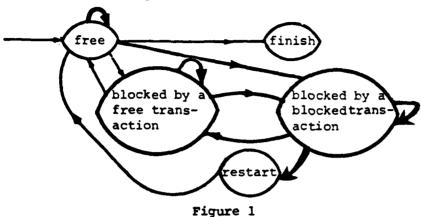
Note that there can be at most one edge emanating from any transaction T because of our assumption that transactions obtain locks sequentially. T is said to be blocked if there is an edge emanating from it; else T is free. New (or restarted) transactions enter the system as free transactions. Whenever a new edge is added to the graph, it must emanate from a previously free transaction (since

free transactions are the only ones that can request locks). We shall assume that the probability of adding more than one edge to the graph in the time interval (t,t + dt) is negligible. In our analysis we assume that deadlock is handled by a deadlock detection algorithm: As soon as a deadlock is formed, an "oracle" detects it and breaks the deadlock cycle instantaneously.

Furthermore, all processing requires finite time. Based on the above assumptions, let us next consider some of the properties of the time-varying waits-for graph. Two properties follow immediately:

Suppose transaction T is blocked by transaction T' at time t. Then (1) if T and T' are both blocked at t and the edge (T,T') does not belong to a cycle, then (T,T') will still be in the waits-for graph at (t + dt) (since processing requires finite time). (2) if T' is free at t, then: either (T,T') will remain in the graph at (t,t + dt) (if T' does not complete at (t,t + dt)) or (T,T') will disappear during (t,t + dt) (if T' completes or restarts during that interval).

From these properties we derive the following state transition diagram (from t to t + dt) for a single transaction



LEMMA 6.1. (1) If we exclude the moments at which cycles are formed and broken by the oracle, then the waits-for graph at time t is a <u>forest</u> of trees, whose roots are the free transactions and the other nodes are the blocked transactions.

(2) A deadlock can be formed at (t,t + dt) only because one root is blocked by one of its descendants.

For the proof of (1) recall that a transaction can only be blocked at *one* data item by *one* transaction. To prove (2), recall our assumption that the probability of adding more than one edge to the graph in (t,t+dt) is negligible.

Next let us consider the trees of the waits-for graph as "nodes" in an evolutionary random graph. Each "node" has a "size" which is the number of transactions in the corresponding tree. The following operations change the graph dynamically from t to t+dt:

- 1. An edge is formed from the root of some tree T_1 to another tree T_2 . $(T_2 \neq T_1)$. Let $\text{size}(T_1) = s_1$ and $\text{size}(T_2) = s_2$. Then the two trees merge to a single tree of size $s_1 + s_2$.
- 2. A new tree of size 1 appears whenever a new transaction enters the system.
- 3. When a cycle is formed from the root of a tree to one of its descendants, then, the oracle immediately restarts some node participating in the cycle. For purposes of this analysis we shall assume that the oracle restarts the root.
- 4. If one root completes, then a number of trees equal to the number of its immediate descendants will be formed.

For further analysis, we make the following assumptions: (i) The service time distribution for each free transaction in a data item is exponential with mean $1/\mu$ (rate μ). (ii) After a local completion, a transaction exits the system with probability θ and requests one more lock with probability $1-\theta$, $0 \le \theta \le 1$. (Note that, according to this, the average number of locks a transaction needs is $1/\theta$). (iii) The probability that a particular root will form an edge with a particular tree at (t,t+dt) is proportional to the size of the tree. This assumption can be justified if we accept that each transaction locks (on the average) the same number of data items and all data items are accessed uniformly.

Let us now define the states of the system:

A state of the evolutionary random graph is an n-tuple $(s_1, ..., s_n)$, where n = number of trees in system at t and s_i = size of tree T_i at t.

Let $p_{ij} = Prob(T_i)$ will conflict with T_j at (t, t + dt) given T_i locally completes and requests another lock).

According to our assumption, $p_{ij} = \alpha \cdot s_j$ for some constant α (depending on current state). The total number of transactions in the system is $s = \sum_{i=1}^n s_i$ when the state is (s_1, \ldots, s_n) . Let M be the total number of granules in the database. The total number of locked places will be approximated here by the average total number of locked granules, a pessimistic estimation of which is equal to $1/\theta \sum_{i=1}^n s_i$. By using the above approximation,

 $Prob(T_i)$ will conflict at (t,t+dt) given it locally completes at $t)+Prob(T_i)$ will not conflict at (t,t+dt) given it locally completes at t) is equal to 1.

Hence,

$$\Sigma_{j=1}^{n} P_{ij} + \frac{M - \frac{1}{\theta} \Sigma_{i=1}^{n} s_{i}}{M} = 1$$

implying $\alpha = 1/\theta M$.

LEMMA 6.2. $P_{ij} = (1/\theta M)s_j$ Vi, Vj. When the system is in state $(s_1, ..., s_n)$ at time t, the overall probability of deadlock will be given by

(1-0) (µdt)
$$\Sigma_{i=1}^{n}$$
 P_{ii}

From this Lemma, we can readily derive:

rate of deadlock =
$$\frac{\mu}{M} \frac{1-\theta}{\theta} \sum_{i=1}^{n} s_i$$

rate of conflicts at t:

$$(1-\theta)\mu \quad \Sigma_{i=1}^{n} \quad \Sigma_{j=1}^{n} \quad P_{ij} = \frac{\mu}{M} \frac{1-\theta}{\theta} \left(\Sigma_{j=1}^{n} s_{j} \right) \cdot n$$

which imply that for state (s_1, s_2, \ldots, s_n)

$$\frac{\text{rate of deadlock}}{\text{rate of conflicts}} = \frac{1}{n}$$
 (Equation (6.1))

(Note that if we want to take into account the fact that a transaction will not cause deadlock with itself then we can use a more refined assumption:

$$p_{ij} = \alpha \cdot s_{j}$$
 if $i \neq j$
 $p_{ij} = \alpha \cdot (s_{j}-1)$ (AR)

Reasoning as above, the value of a will now be

$$\alpha = \frac{1}{\theta M} \cdot \frac{s}{s-1}$$
 where $s = \sum_{i=1}^{n} s_i$

For large s, $\alpha \simeq 1/\theta M$ again. Approximating then α by $1/\theta M$ we finally get

rate of deadlock =
$$\frac{\mu}{M} = \frac{1-\theta}{\theta}$$
 · s

rate of conflicts at
$$t = \frac{\mu}{M} \frac{1-\theta}{\theta}$$
 · (s-1) · n

which imply that for state (s_1, s_2, \dots, s_n)

$$\frac{\text{rate of deadlock}}{\text{rate of conflicts}} = \frac{s}{(s-1) \cdot n}$$
 (Eq. 6.1B)

which is $\simeq 1/n$ for large s.)

For reasons of clarity we use the "crude" assumption $(p_{ij} = \alpha \cdot s_j)$ Vi Vj) in the analysis following.

We now consider the probabilities of the possible state transitions of the system (from time t to t+dt).

(1) An arrival

$$(s_1, ..., s_n)$$
 $(s_1, ..., s_n, 1)$
with prob = $\lambda \cdot dt$

(2) A local completion and a conflict (not a deadlock), for n > 1

with probability (μdt) ($1-\theta$) • $\frac{s_j}{\theta M} = \mu \frac{s_j}{M} dt \frac{(1-\theta)}{\theta}$

(3) A deadlock (the root in a tree completes locally and then conflicts with its own descendants).

$$\forall j, 1 \le j \le n \quad \forall m = 0,...,s_{j} - 1$$

$$(s_{1},...,s_{j},...,s_{n}) \quad (s_{1},...,s_{j-1}, (s_{j_{1}},...,s_{j_{m}}),...,s_{n},1)$$

such that

$$j_1 + s_{j_2} + \cdots + s_{j_m} = s_{j_m} - 1$$
 and $s_{j_m} > 0$

The probability of this transition is

Obviously, m is the number of immediate siblings of the root of T_{j} just before the deadlock.

LEMMA 6.3: The $Prob\{m | deadlock in j\} = Prob(the root of T_j has m immediate descendants at time of deadlock) can be approximated by$

$$\frac{R(m,s_{j})}{N(s_{j})} \qquad \text{for all } m = 1,...,s_{j} - 1$$

where

and

R(m,s,) = number of trees (labelled) of s, nodes and a distinct specified root whose degree is m

 $N(s_j)$ = number of labelled trees of s_j nodes and a distinct specified root, $(1 \le degree(root) \le s_j - 1)$

such that the following hold for $R(m,s_i)$ and $N(s_i)$:

(1)
$$N(k) = k^{k-2}, k > 2$$

(II)
$$R(m,k) = {k-1 \choose m} \sum_{\forall (k_1,\ldots,k_m)}$$

such that $k_1 + \ldots + k_m = k-m-1$

$$\begin{bmatrix} \binom{n-m-1}{k_1}^{(k_1+1)}^{(k_1+1)} & \binom{n-m-1-k_1}{k_2}^{(k_2+1)}^{(k_2+1)}^{(k_2+1)}^{(k_2+1)} & \cdots & \binom{k_m}{k_m}^{(k_m+1)}^{(k_m+1)} \end{bmatrix}$$

with $1 \le m \le k - 1$

(III) N(1) = N(0) = 1 by definition.

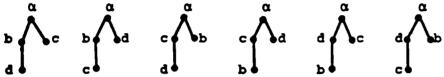
<u>Proof.</u> Assuming uniformity in the way trees merge during conflicts, at steady state, $Prob\{m \mid deadlock\}$ will indeed be equal to

N(k) by definition is the number of rooted labelled trees of $k \ge 2$ nodes and can be shown to be equal to k^{k-2} [Ca, 89]. Note that the trees are labelled since we have distinct transactions as nodes, and that the class of trees which define the same partial order (with the ordering $i < j \iff i$ is son of j) must be counted as only one tree.

To form R(m,k), $1 \le m \le k-1$, m sons of the root are to be selected from the k-1 nodes in

$$\binom{k-1}{m}$$

ways, and for each of them, partition the remaining k-l-m nodes in m groups (group i has k, nodes, $0 \le k, \le k-m-1$, $1 \le i \le m$). Then count all possible ways of selecting labels for each of these m groups and for each k, and one of its labellings, we have to count all possible (labelled) rooted trees formed with k, +l nodes.



In order to calculate the transition probability due to deadlock, it is necessary to consider the $\operatorname{prob}\{s_{j_1},\ldots,s_{j_m}|m \text{ and deadlock in }s_j\}$. When deadlock occurs in the jth tree, the s_j^2-1 descendants will be fragmented into m new trees $0 \le m \le s_j-1$. This process is analogous to grouping s_j-1 into m random partitions. The probability of a certain partition is [Fe, 66]

$$Prob\{s_{j_{2}},...,s_{j_{m}}|_{m,s_{j}}\} = \frac{(s_{j}^{-1})!}{s_{j_{1}}!s_{j_{2}}!...s_{j_{m}}!} \cdot \left(\frac{1}{m}\right)^{s_{j}^{-1}}$$

We, therefore, conclude that the transition due to deadlock,

$$(s_1,\ldots,s_j,\ldots,s_n) \xrightarrow{} (s_1,\ldots,s_{j-1},(s_{j_1},\ldots,s_{j_m}),\ldots,s_n,1)$$

has probability

$$\frac{1-\theta}{\theta} \; (\mu dt) \cdot \left(\frac{s_{j}}{M}\right) \cdot \frac{R(m,s_{j})}{N(s_{j})} \; \cdot \left[\frac{(s_{j}-1)!}{s_{j_{1}}! \dots s_{j_{m}}!} \left(\frac{1}{m}\right)^{s_{j}-1}\right]$$

(4) A departure from the system. (A root totally completes and departs.)

$$\forall j \quad \forall m = 0,1,...,s_{j} -1$$

$$(s_{1},...,s_{j},...,s_{n}) \longrightarrow (s_{1},...,s_{j-1},(s_{j_{1}},...,s_{j_{m}}),s_{j+1},...,s_{n})$$

(where the root of T, completes). By the same reasoning as in the case of dead-lock, this transition has probability

$$(\mu dt) \cdot \theta \cdot \frac{R(m,s_j)}{N(s_j)} \cdot \left[\frac{(s_j-1)!}{s_{j_1}! \dots s_{j_m}!} \left(\frac{1}{m} \right)^{s_j-1} \right]$$

where

$$s_{j_1} + s_{j_2} + \dots + s_{j_m} = s_{j-1}$$

On identifying all the transition probabilities, the process is now completely defined. Due to the complicated transitions arising in cases of deadlock and total completion, in general the process does not necessarily observe the one-step behavior as defined in [D-B, 78]. Consequently, it is conjectured that the closed form solution for the steady state probabilities is unlikely to be product form as defined in [BCMP, 76]. Nevertheless one can obtain numerical solutions for the problem, should the closed form solution fail to exist.

From the steady state solution, we can now compute the conflict rate \mathbf{r}_{c} and dead-lock rate \mathbf{r}_{d} at steady state

$$\mathbf{r_{c}} = \frac{\mu}{M} \left[\sqrt{\sum_{s \text{ tates}}} \left(\mathbf{n} \cdot \left(\sum_{j=1}^{n} \right) \mathbf{s_{j}} \cdot \operatorname{prob}(\mathbf{s_{1}}, \dots, \mathbf{s_{n}}) \right) \right] \frac{1-\theta}{\theta}$$

$$\mathbf{r_{d}} = \frac{\mu}{M} \cdot \left[\sqrt{\sum_{s \text{ tates}}} \left(\left(\sum_{j=1}^{n} \right) \mathbf{s_{j}} \cdot \operatorname{prob}(\mathbf{s_{1}}, \dots, \mathbf{s_{n}}) \right) \right] \frac{1-\theta}{\theta}$$

(remark $r_d = r_c \cdot 1/n$ for all states in state space.) Note that r_c and r_d only require the aggregate joint probability,

$$f(n,s) = Prob \left(\sum_{i=1}^{n} s_{i} = s \right)$$

where

and

n = number of free transactions in system

s = number of transactions in system.

From the above, we conclude

LEMMA 6.4.

$$r_c = \frac{\mu}{M} E(n \cdot s) \frac{1-\theta}{\theta}$$
 and $r_d = \frac{\mu}{M} \cdot E(s) \cdot \frac{1-\theta}{\theta}$

or the rate of deadlocks at steady state is proportional to the average number of transactions in the system, and the rate of conflicts is proportional to the mean value of the product $n \cdot s$ where n = number of free transactions in the system and <math>s = total number of transactions in the system.

(Note that, by using the more refined assumption AR we would get

$$\mathbf{r_{d}} = \frac{\mu}{M} \frac{1-\theta}{\theta} \left[\mathbf{E}(\mathbf{s}) - \mathbf{E}(\mathbf{n}) \right]$$

$$\mathbf{r_{c}} = \frac{\mu}{M} \frac{1-\theta}{\theta} \left[\mathbf{E}(\mathbf{n} \cdot \mathbf{s}) - \mathbf{E}(\mathbf{n}) \right] .$$

Note that the results of this lemma contradict the assumption in [R, 79] that the multiprogramming level has no affect on the performance in 2PL systems.

6.1 SIMULATION ANALYSIS OF THE 2PL ALGORITHMS

Recently, some simulation studies are performed to analyze the performance of the 2PL methods. [L-N, 81]. The overall system model is very similar to the model here as described in section 5. There are more general assumptions in the simulation which include the Erlangian service times at the granules and constant times for the hops. The study focused on larga databases (3000 \leq M \leq 12000) and simulations are carried out for both the "simple" and general 2PL. Their major results agree very favorably to our results. For example:

- (1) The number of transactions in the system affects the overall performance (thereby contradicting Ries [R, 79] and agreeing with the results of lemma 6.4.
- (2) The number of transactions under deadlock increases almost linearly with the number of locks held by the active transactions and the multiprogramming level, validating our results of equations 6.1 and lemma 6.4 (r proportional to 1 E(s)).
- (3) The rate of conflicts increases more than linearly with the multiprogramming level (close to quadratic in heavy traffic) validating our second result of lemma 6.4 (r proportional to $\frac{1}{6}$ E(s.n)).
- (4) Similar locable unitls imply a smaller probability of conflict, as we expect from the analysis of section 5.

7.0 CONCLUSION

We developed an analytic model for the general 2PL, which can be used to estimate the stead state rates of conflicts and deadlocks in the system. Under reasonable assumptions for transaction behavior, we found that the rate of deadlocks is proportional to the average number of transactions in the system and the rate of conflicts is proportional to the mean of the product of the number of free transactions in system multiplied by the total number of transactions in system. Since the rate of deadlocks is equal to the rate of restarts and because restarts affect directly the number of transactions in the system, continued analysis is important for further understanding and quantification of the system performance under this feedback effect. The next step in our research will be a system model using decomposition and the results obtained above.

REFERENCES

- [1] [BCMP, 75] Baskett, F., Chandy, K.M., Muntz, R.R., and Palacios, J., Open, closed and mixed networks of queues with different classes of customers, JACM 22, 2 (1975) 248-260.
- [2] [B-G, 78] Bernstein, P., and N. Goodman, "Approaches to concurrency control in distributed database systems," Harvard University, Center for Research in Computing Technology, TR-26-78, 1978.
- [3] [B-G, 80] Bernstein, P., and Goodman, A., Fundamental algorithms for concurrency control in distributed database systems, CCA TR Contract No. F30603-79-C-0191, Cambridge, MA (1980).
- [4] [BSW, 79] Bernstein, P., Shipman, D., and Wong, W., Formal aspects of serializability in database concurrency control, IEEE Trans. Softw. Eng. SE-5,3 (]979) 203-215.
- [5] [Ca, 89] Cayley, A., A theorem on trees, Quart. J. Math 23 (1889) 376-378.

 Mathematical Papers, Cambridge, England, 13 (1897) 26-28.
- [6] [D-B, 78] Denning, P.J., and Buzen, J.P., The operational analysis of queuing network models, ACM Comp Surveys 10,3 (1978) 225-261.
- [7] [Fe, 66] Feller, W., An Introduction to Probability Theory and Its Applications, Vol. I (Wiley, New York, 1966).
- [8] [Ge-S, 78] Gelenbe, E., and Sevcik, K., Analysis of update synchronization for multiple copy databases, Proc. 3rd Berkeley Workshop on Distributed Databases and Computer Networks (August, 1978).
- [9] [GM, 79] Garcia-Molina, H., Performance of Update Algorithms for Replicated Data in a Distributed Database, Ph.D. Thesis, Computer Science Dept., Stanford Univ. (June, 1975).
- [10] [Ha, 69] Harary, F., Graph Theory, (Addison-Wesley Series in Mathematics, 1969).
- [11] [Klei, 76] Kleinrock, L., Queuing Systems, Vol. I (Wiley, New York, 1976).
- [12] (L-N, 81] Lin, K and J. Nolte, "Performance of 2PL," CCA TR, 1981
- [13] [M-N, 79] Menasce, D., and Nakanishi, T., Optimistic Versus Pessimistic Concurrency Control Mechanisms in Database Management Systems, Dept. de Informatica, Pontificia Universidade Catolica do Rio de Janeiro, Brasil (1979).
- [14] [P-L, 80] Potier, D., and Leblanc, P.L., Analysis of locking policies in database management systems, CACM 23,10 (1980) 584-593.
- [15] [R, 79] Ries, D., The Effect of Concurrency Control on Database Management System Performance, Ph.D. Thesis, Computer Science Dept., Univ. of California, Berkeley (April, 1979).

- [16] [RLS, 78] Rosenkrantz, D.J., Stearns, R.E., and Lewis, D.M., System level concurrency control for distributed database systems, ACM Transactions on Database Systems 3,2 (June, 1978) 178-198.
- [17] [S-78] Stewart, W.G., A comparison of numerical techniques in Markov modelling, CACM (Feb. 1978).
- [18] [Th, 79] Thomas, R.H., A majority consensus approach to concurrency control for multiple copy databases, ACM Transactions on Database Systems 4,2 (1979) 180-209.
- [19] [Th, 78] Thomas, R.H., A solution to the concurrency control problem for multiple copy databases, Proc. 1978 COMPCON Conference, IEEE, New York (1978).

SECTION V

A SIMPLE ANALYTIC MODEL FOR PERFORMANCE OF EXCLUSIVE LOCKING IN DATABASE SYSTEMS*

Nathan Goodman

Rajan Suri

Yong C. Tay

^{*}To appear in the Proceedings of the Second ACM SIGACT-SIGMOD Symposium on Principles of Database Systems, March 1983.

1. INTRODUCTION

Concurrency control in database systems is the coordination of concurrent access to shared data. Much is already known about the design of concurrency control algorithms in both centralized and distributed systems; the majority of these algorithms are in fact just combinations of two basic techniques—locking and timestamp ordering [BG], and there is now considerable interest in understanding the performance of these two techniques.

In analyzing any system, it is frequently true that an appropriate model can provide better insight than a simulation study, and at less cost. To date, most performance analyses have been done either experimentally [LN1,LN2,MK,R] or with Markovian queueing-theoretic means [L,G,PL,SS].

This paper introduces a new framework for analyzing exclusive locking that uses none of the usual assumptions of queueing theory besides Little's Law [K]. For example, we do not assume that certain quantities are exponentially distributed. The model presented is easy to understand and costs little to solve computationally, yet captures the essential features of the system it models.

The new framework is presented in Section 2 in a general form. Then, in Section 3, we use the framework to consider the case where there is no blocking. The resulting model is applied to analyze three different systems in Sections 4, 6 and 7. Section 5 presents simulation results that validate the model for the case considered in Section 4. Conclusions about the systems studied and the model itself are in Section 8.

2. THE FRAMEWORK

The database is a collection of data items. Each transaction makes a sequence of requests; the time between the i-th and (i+1)-th request is the same for all transactions. The 0-th request is a request to start (immediately granted), and the i-th request $(i \ge 1)$ is for either an exclusive lock on a data item, or termination. Let $prob(i-th request is for termination) = p_i$, $p_0 = 0$.

When a transaction makes a lock request, it is immediately granted if there is no conflict; otherwise, the transaction either waits or is restarted. Let prob(conflict) = p. Requests for termination are granted immediately, and each termination starts another transaction (so the number of transactions is constant). A transaction releases its locks if and only if it is restarted or terminated.

The model for this system consists of two parts. The first is the flow diagram illustrated in Figure 2.1.

The second part of the model is a set of equations describing the behavior of the system. These equations are derived using the steady state average values of the variables. The underlying idea in our approach is to characterize the system in terms of these average values, instead of detailed dynamics involving instantaneous values of each variable.* The input parameters to the model are:

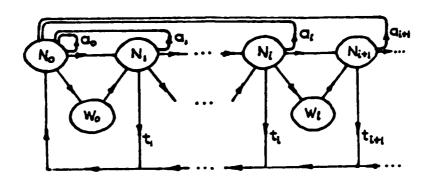
N - the number of transactions in the system.

D - the number of data items; this specifies the granularity.

 p_i - the probability of termination with i locks, i = 1, 2, ...

It was pointed out to the authors by Ken Sevcik that a related approach has been proposed for large multiclass queueing models [LZ].

Figure 2.1. Flow Diagram for the Model.



 N_i is the number of transactions with i locks that are executing W_i is the number of transactions with i locks that are waiting a_i is the abort rate of transactions holding i locks t_i is the rate of termination of transactions holding i locks, $t_0 = 0$. We refer to each (N_i) as a stage. The time T between the i-th and (i+1)-th requests is assumed to be a function of i, the number of executing transactions $N_e = \sum_{j=0}^{\infty} N_j$, and the number of waiting transactions $N_w = \sum_{j=0}^{\infty} W_j$.

d - a function of i which is a measure of the interval (such as the number of instructions) between the i-th and (i+1)-th requests.

T — a function of N_e and N_w spacifying the time taken per unit measure of the inter-request interval (hence $T(i,N_e,N_w) = d(i)T(N_e,N_w)$); T reflects the effect of multiprogramming on the rate of execution.

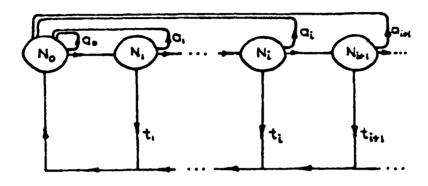
 π — a function of D, $N = (N_0, N_1, \ldots)$ and $W = (W_0, W_1, \ldots)$ characterizing the conflicts; it specifies the dependence of the conflict probability p on D, N and W, i.e. $p = \pi(D, N, W)$. Given these inputs, the task is to determine N and W, and hence deduce the values of various performance measures.

In this paper, we shall restrict our attention to the case where no waiting is allowed. There are two reasons for this: Firstly, it is easier to illustrate the approach in our framework. Secondly, even this simple case provides some useful insight, and it will guide us in analyzing the waiting case.

3. THE NO WAITING CASE

In this case, a transaction requesting a lock is restarted whenever there is a conflict. The flow diagram has the simplified form in Figure 3.1.

Figure 3.1. The No Waiting Case



For convenience let d(i) be independent of i, say d(i) = 1 for all i.

Then, since $N_e = N$ and $N_w = 0$ in this model,

(Inter-Request Time)
$$T = T(N)$$
. (3.1)

Using Little's Law, $N_i = r_i T$ in steady state, where r_i is the rate at which transactions enter stage i. Hence, from Fig. 3.1,

(Little's Law)
$$N_i = r_i T$$
 where $r_i = \sum_{j=i}^{\infty} t_j + \sum_{j=i}^{\infty} a_j$ $i=0,1,2,...$ (3.2)

Since the number of transactions in the system is held constant, we have

(Conservation)
$$N = \sum_{j=0}^{\infty} N_{j}$$
 (3.3)

Now, $p_i = t_i/r_i$ or $t_i = p_i r_i$. By (3.2),

(Rate of Termination)
$$t_i = \frac{1}{T} p_i N_i$$
 $i=1,2,...$ (3.4)

Similarly, since at each stage, the rate of lock requests is $(1-p_i)r_i$, so $p = \frac{a_i}{(1-p_i)r_i}$, and thus

(Abort Rate)
$$a_i = \frac{1}{T} p(1-p_i) N_i \qquad i=0,1,2,... \quad (3.5)$$

Since we are considering the case where there is no waiting for a lock,

(Probability of Conflict)
$$p = \pi(D, N)$$
 . (3.6)

For arbitrary π and τ , p--and hence N--can be evaluated by solving equations (3.1)-(3.6) (for example, by iteration).

To continue with our analysis, we shall henceforth assume there is uniform access,* i.e.

$$\pi(D,N) = \frac{1}{D} \sum_{j=1}^{\infty} jN_{j} .**$$
 (3.7)

Fact 3.1: The probability of conflict p is a solution to f(q) = 0, where

$$f(q) = \frac{\lambda}{(1-q)} \frac{\sum_{j=1}^{\infty} j c_j q^j}{\sum_{j=0}^{\infty} c_j q^j} - 1$$
(3.8)

$$q = 1-p$$
, $\lambda = \frac{N}{D}$, $c_0 = 1$ and $c_i = \frac{i-1}{n} (1-p_i)$ for $i=1,2,...$

If one assumes that a transaction does not request for a lock it already holds, then this expression should in fact be $\binom{\infty}{\Sigma} j N_j - i / (D-i)$ for a transaction with i locks. The two expressions converge if $i << \sum_{j=1}^{\infty} j N_j$.

[[]TSG] considers non-uniform access.

<u>Proof</u>. By (3.2),

$$\frac{N_{i+1}}{N_i} = \frac{r_{i+1}}{r_i} = 1 - \frac{t_i + a_i}{r_i} = 1 - \frac{p_i r_i + p(1-p_i) r_i}{r_i} = (1-p)(1-p_i) .$$

Hence

$$N_{i+1} = q(1-p_i)N_i$$

or

$$N_i = N_0 \prod_{j=0}^{i-1} [q(1-p_j)] = c_i q^i N_0$$
 for i=0,1,... (3.9)

Using (3.3)

$$N = \sum_{j=0}^{\infty} c_j q^j N_o,$$

50

$$N_0 = \frac{N}{\infty}$$

$$\sum_{j=0}^{\infty} c_j q^j$$
(3.10)

Now, by (3.7)

$$p = \frac{1}{D} \sum_{j=1}^{\infty} j c_{j} q^{j} N_{0} = \frac{N}{D} \cdot \frac{N_{0}}{N} \cdot \sum_{j=1}^{\infty} j c_{j} q^{j}$$

or

$$(1-q) = \lambda \frac{\sum_{j=1}^{\infty} jc_{j}q^{j}}{\sum_{j=0}^{\infty} c_{j}q^{j}}$$

Hence the claim.

Remark: For uniform access, the effects of N and D are expressed — through a single parameter $\lambda = \frac{N}{D}$, which we call the *load*.

The following fact indicates that f is well formed.

Fact 3.2: Given λ and c_1 , j=1,2,..., f has exactly one zero in [0,1].

Proof. Let
$$g(q) = \sum_{j=1}^{\infty} jc_j q^j$$
 and

$$h(q) = (1-q) \sum_{j=0}^{\infty} c_j q^j = \sum_{j=0}^{\infty} c_j q^j - \sum_{j=0}^{\infty} c_j q^{j+1} = 1 - \sum_{j=1}^{\infty} (c_{j-1} - c_j) q^j.$$

Since $c_j = (1-p_j)c_{j-1} \le c_{j-1}$, we have $c_{j-1} - c_j \ge 0$ and so h(q) decreases as q increases, whereas g(q) increases as q increases. But $f(q) = \lambda \frac{g(q)}{h(q)} - 1$, so f(q) increases as q increases. The claim now follows from the observation that $f(0) = -1 \le 0$, $\lim_{q \to 1^-} f(q) = +\infty$ and f is continuous on $q \to 1$.

Furthermore, this zero behaves in a manner expected of it.

Fact 3.3: Given c_j , j=1,2,..., prob(conflict) increases as λ increases, and $\lim_{\lambda \to 0^+} p = 0$, $\lim_{\lambda \to +\infty} p = 1$.

<u>Proof.</u> For fixed q, f(q) increases as λ increases. Hence the zero q_{λ} in [0,1) must decrease as λ increases since f is monotonic increasing on [0,1). (See Fact 3.2 and Figure 3.2). Since $p=1-q_{\lambda}$, p therefore increases as λ increases. Now

$$0 < \lambda \sum_{j=1}^{\infty} j c_j q_{\lambda}^j < \lambda \sum_{j=1}^{\infty} j c_j, \quad \text{so} \quad \lim_{\lambda \to 0^+} \lambda \sum_{j=1}^{\infty} j c_j q_{\lambda}^j = 0, \quad \text{and}$$

$$\sum_{j=0}^{\infty} c_j q_{\lambda}^j = 1 + \sum_{j=1}^{\infty} c_j q_{\lambda}^j > 1.$$

Since $f(q_{\lambda}) = 0$, we have

$$0 < (1-q_{\lambda}) = \frac{\lambda \sum_{j=1}^{\infty} j c_{j} q_{\lambda}^{j}}{\sum_{j=0}^{\infty} c_{j} q_{\lambda}^{j}} < \lambda \sum_{j=1}^{\infty} j c_{j} q_{\lambda}^{j}$$

Therefore

$$0 \le \lim_{\lambda \to 0^+} (1-q_{\lambda}) \le \lim_{\lambda \to 0^+} \lambda \sum_{j=1}^{\infty} jc_j q_{\lambda}^j = 0, \quad \text{i.e. } \lim_{\lambda \to 0^+} p = 0.$$

Suppose $\lim_{\lambda\to +\infty} p<1$. (The limit exists since p increases with λ and is bounded above by 1.) Then $\lim_{\lambda\to +\infty} q_{\lambda}=\epsilon$ for some $0<\epsilon \le 1$, and $\lambda\to +\infty$

$$\lim_{\lambda \to +\infty} f(q_{\lambda}) = \lim_{\lambda \to +\infty} \frac{\lambda \sum_{j=1}^{\infty} j c_{j} \epsilon^{j} - (1-\epsilon) \sum_{j=0}^{\infty} c_{j} \epsilon^{j}}{\sum_{j=0}^{\infty} c_{j} \epsilon^{j}}$$

$$= \lim_{\lambda \to +\infty} \frac{1}{\sum_{j=0}^{\infty} c_{j} \epsilon^{j}} \left[\lambda \sum_{j=1}^{\infty} j c_{j} \epsilon^{j} - 1 + \sum_{j=1}^{\infty} (c_{j-1} - c_{j}) \epsilon^{j} \right]$$

$$= \lim_{\lambda \to \infty} \frac{1}{\sum_{j=0}^{\infty} c_{j} \epsilon^{j}} \left[\lambda \sum_{j=1}^{\infty} j c_{j} \epsilon^{j} - 1 \right]$$
(see proof of Fact 3.2)

= $+\infty$, which contradicts the definition of q_1 .

Hence lim p=1.

Using the zero q_{λ} , one can compute N using (3.9) and (3.10). From (3.4), the throughput is then

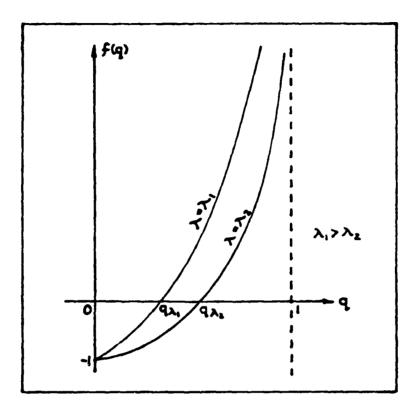
$$t = \sum_{j=1}^{\infty} t_j = \frac{1}{T} \sum_{j=1}^{\infty} p_j N_j$$
, (3.11)

and, from (3.5), the abort rate is

$$a = \sum_{j=0}^{\infty} a_{j} = \frac{p}{T} \left(\sum_{j=0}^{\infty} N_{j} - \sum_{j=0}^{\infty} p_{j} N_{j} \right) = \frac{p}{T} \left(N - \sum_{j=0}^{\infty} p_{j} N_{j} \right) . \tag{3.12}$$

Other performance measures like the number of locks a transaction holds when it restarts/terminates, response time of a transaction and prob(transaction terminates without ever restarting) can also be computed.

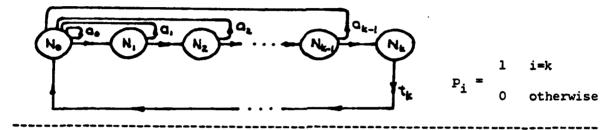
Figure 3.2 $\, q_{\hat{\lambda}} \,$ decreases as $\, \lambda \,$ increases.



4. CASE WHERE ALL TRANSACTIONS REQUIRE & LOCKS

We first apply the model to a system where all transactions require k locks to complete. In this case, the flow diagram has the simplified form in Figure 4.1.

Figure 4.1. All transactions require k locks.



As for the equations, we have, from (3.8),

$$c_{i} = \begin{cases} 0 & i > k \\ 1 & 0 \le i \le k \end{cases} \tag{4.1}$$

and hence

$$f(q) = \frac{\lambda}{1-q} \frac{\sum_{j=1}^{k} jq^{j}}{\sum_{j=0}^{k} q^{j}} - 1$$

$$= \frac{\lambda}{1-q^{k+1}} (kq^{k} + (k-1)q^{k-1} + \dots + q) - 1 . \qquad (4.2)$$

Equations (3.9), (3.10) and (4.1) now give $N_i = 0$ for i > k and

$$N_i = \frac{Nq^i}{1+q+...+q^k} = \frac{N(1-q)q^i}{1-q^{k+1}}$$
 for i=0,1,...,k. (4.3)

From (3.11) and (3.12), the throughput and abort rates are

$$t = \frac{1}{T} N_k = \frac{N}{T} \frac{(1-q)q^k}{1-q^{k+1}}$$
 (4.4)

and

$$a = \frac{p}{T} (N - N_{k})$$

$$= \frac{(1-q)}{T} \left[N - \frac{N(1-q)q^{k}}{1-q^{k+1}} \right]$$

$$= \frac{N}{T} \frac{(1-q)(1-q^{k})}{1-q^{k+1}} . \qquad (4.5)$$

The zero in [0,1) of (4.2) is found with the help of MACSYMA* for various values of λ . We shall compare the numerical solution to some simulation results in the next section. First, let us analyze the asymptotic behavior of the system as predicted by these formulae.

Fact 4.1: For small λ and k (i.e. $\lambda << 1$ and k < 10 say)

(i)
$$p = \frac{kN}{2D}$$
 (ii) $t = \frac{N}{(k+1)T}$ (iii) $a = kpt$

<u>Proof.</u> By Fact 3.3, $q \approx 1$ for $\lambda \approx 0$, and hence $q^k \approx 1$ for small k. Using this approximation on (4.2),

$$0 = f(q) \approx \frac{\lambda}{p} = \frac{\sum_{j=1}^{k} j}{k} - 1 = \frac{\lambda k}{2p} - 1$$

$$\sum_{j=0}^{k} 1$$

50

$$p = \frac{\lambda k}{2} = \frac{k}{2} \frac{N}{D} .$$

MACSYMA is a large symbolic manipulation program developed at the MIT Laboratory for Computer Science and supported by the National Aeronautics and Space Administration under grant NSG 1323, by the Office of Naval Research under grant N00014-77-C-0641, by the U.S. Department of Energy under grant ET-78-C-02-4687, and by the U.S. Air Force under grant F49620-79-C-020.

For throughput, by (4.4),

$$t = \frac{N}{T} \frac{q^k}{1+q+\dots+q^k} \approx \frac{N}{(k+1)T}.$$

Now, (4.4) and (4.5) implies
$$\frac{a}{t} = \frac{1-q^k}{q^k} = \frac{1}{q^k} - 1$$
, so

$$a = ((1-p)^{-k}-1)t$$

≈ kpt

since $p \approx 0$ and k is small.

Hence, for small λ and k, p and t are linear in N. Consequently, a is quadratic in N (but a/t is linear) for small λ and k.

Graph 4.1 compares these asymptotes with exact solutions obtained by solving for p using MACSYMA.

Fact 4.2:

$$\lim_{k\to\infty} p = \frac{1}{2} \left[\sqrt{\lambda^2 + 4\lambda} - \lambda \right] .$$

Proof. Since $\lim_{k\to\infty} q^{k+1} = 0$ and $\frac{q}{(1-q)^2} = q + 2q^2 + \cdots + kq^k + \cdots$,

$$\lim_{k\to\infty} f(q) = \frac{\lambda q}{(1-q)^2} - 1 \stackrel{\triangle}{=} f_{\infty}(q) \quad \text{say.}$$

The zeros of f_{∞} are given by

$$(1-q)^2 - \lambda q = 0$$
 or $q^2 - (2+\lambda)q + 1 = 0$.

They are

$$q = \frac{1}{2} \left[(2+\lambda) \pm \sqrt{(2+\lambda)^2 - 4} \right]$$

The zero in (0,1) is $q = \frac{1}{2} \left[2+\lambda - \lambda \sqrt{\lambda^2 + 4\lambda} \right]$, so $p = 1 - q = \frac{1}{2} \left[\sqrt{\lambda^2 + 4\lambda} - \lambda \right]$.

Graph 4.2 shows how this limit is reached for various λ .

We now discuss the significance of these results.

Fact 4.1 confirms our intuition that, for small λ , the transactions hardly interfere (i.e., seldom conflict) with one another. For, with little interference, almost all transactions are expected to terminate without restarts. The average number of locks held by each transactions is then $\frac{k}{2}$, so that there are $\frac{k}{2}$ N locks in the system, giving $p = \frac{1}{D} (\frac{k}{2} N)$. Also, the response time is (k+1)T since there are k+1 stages; hence $t = \frac{N}{(k+1)T}$. Finally, if there is scant interference, then the throughput through each stage is t, the abort rate at each of the first k stages is pt, and so a = kpt. Note that we have derived these three approximations simply by assuming there is little conflict of locks, but they agree with the approximations in Fact 4.1, which are obtained by examining the solution to f(q) = 0.

Fact 4.2 and Graph 4.2 show that prob(conflict) is constant for large enough k. Given λ_0 , call the least k for which prob(conflict) is within 1% of the limiting value the lock limit for λ_0 . That prob(conflict) is constant must mean that increasing the number (k) of stages makes no difference to the number of locks held. This, in turn, must be because the number of transactions at the last stage is negligible. Thus, for a given number of stages k_0 , call the value of λ for which $N_k/N=1$ % the saturation point for k_0 . We expect that, for a given λ_0 , if k is larger than its lock limit, then λ_0 must be close to, if not larger than the saturation point for k. For example, for $\lambda_0=0.9$, the lock limit is 5 (see Graph 4.2), and the saturation point for k=5 is between 0.5 and 0.9 (see Graph 4.3).

The model also captures the phenomenon of thrashing due to concurrency control and resource contention. Graph 4.4a is a plot of t for T(N) = 1

(i.e. no resource contention). We see that t decreases for large N due to the restarts caused by the concurrency control. Call the maxima the CC-thrashing point. On the other hand, suppose there is no concurrency control. A typical T is an approximately exponential function that gives a throughput curve like that in Graph 4.4b [DKLPS]. Again, t decreases for large N, but this is the effect of resource contention—call the maxima the RC-thrashing point. Graph 4.4c shows the combined effect of concurrency control and resource contention.

For a given k, one would expect the CC-thrashing point to be reached even before the saturation point: From Graph 4.4a, the CC-thrashing point for k=5 is at $\lambda=0.25$, whereas the saturation point is greater than 0.5 (Graph 4.3).

5. VALIDATION OF THE MODEL

The values of p as a function of k and λ , and as computed by MACSYMA using (4.2), are tabulated in Table 5.1. Tables 5.2 to 5.6 give various performance measures computed with these values.

To validate the model, we use a restricted version of the two phase locking simulation program written by Oded Shmueli. In the simulation runs, D=100, k varies from 3 to 6, and λ from 0.1 to 0.9. In each run, 30000 requests are generated, and the average performance measures computed for the last 10000 requests (to avoid transient effects).

Table 5.7 and Graph 5.1 compare the simulation results with the figures predicted by the model. The agreement is remarkable. Note from the table that the ratio a/t varies from 0.5 to 230.0, so we have simulated a high level of conflicts in some of the experiments. And it is precisely in a situation of intense conflict that we expect deficiencies in a model to show temselves. For this reason, the results in Table 5.1 are reassuring.

A scrutiny of Tables 5.3, 5.5 and 5.6 reveals something rather curious. By fitting the data with a polynomial, one can estimate from Table 5.3 the CC-thrashing points for various k. Using these estimates, one can find the corresponding r and s values in Tables 5.5 and 5.6 by interpolation (again fitting the data with a polynomial).

Table 5.8 shows the results of this exercise: at the CC-thrashing point for a wide range of values of k, r is about 7 and s about 0.3. Why this is so is not yet known. However, from the value of r, one sees that, to achieve maximum throughput, one would have to accept the cost of 7 aborts per successful completion. Looking at it in another way, to reach the CC-thrashing point, one would have to push the system very hard.

6. TRANSACTIONS WITH MEMORYLESS BEHAVIOR

Suppose after acquiring each lock, a transaction terminates with probability θ (it does not 'remember' how many locks it already has). In this case (the flow diagram is as in Figure 3.1), $p_i = \theta$ for $i \ge 1$. From (3.8), we have

$$f(q) = \frac{\lambda}{(1-q)} \frac{\sum_{j=1}^{\infty} j(1-\theta)^{j-1}q^{j}}{\sum_{j=1}^{\infty} (1-\theta)^{j-1}q} - 1 \quad \text{since } c_{0} = 1 \text{ and } c_{j} = (1-\theta)^{j-1} \text{ for } j \ge 1$$

$$= \frac{\lambda}{(1-q)} \frac{q/(1-(1-\theta)q)^{2}}{1+q/(1-(1-\theta)q)} - 1$$

$$= \frac{\lambda q}{(1-q)(1+\theta q)(1-(1-\theta)q)} - 1 \quad . \quad (6.1)$$

As for throughput,

$$t = \frac{1}{T} \sum_{j=1}^{\infty} \theta N_{j} \qquad \text{by (3.11)}$$

$$= \frac{\theta}{T} \sum_{j=1}^{\infty} c_{j} q^{j} N_{0} \qquad \text{by (3.9)}$$

$$= \frac{\theta}{T} N \frac{\sum_{j=1}^{\infty} c_{j} q^{j}}{\sum_{j=0}^{\infty} c_{j} q^{j}} \qquad \text{by (3.10)}$$

$$= \frac{\theta}{T} N \left[\frac{q/(1-(1-\theta)q)}{1+q/(1-(1-\theta)q)} \right]$$

$$= \frac{N}{T} \theta \left[\frac{q}{1+\theta q} \right] \qquad (6.2)$$

Fact 6.1: If θ , T and D are constants, then t increases monotonically with N for large N.

Proof. Recall from Fact 3.3 that $\lim_{\lambda\to +\infty} p=1$ or $\lim_{\lambda\to +\infty} q=0$. Hence for fixed $\lim_{\lambda\to +\infty} p=1$ or $\lim_{\lambda\to +\infty} q=0$. Hence for fixed $\lim_{\lambda\to +\infty} p=1$ or $\lim_{\lambda\to +\infty} q=0$.

$$0 = f(q) = \frac{\lambda q}{(1-q)(1+\theta q)(1-(1-\theta)q)} - 1$$

$$\approx \frac{\lambda q}{1-2(1-\theta)q} - 1$$

SO

$$q \approx \frac{1}{\lambda + 2(1-\xi)}$$

Substituting this into (6.2)

$$t \approx \frac{N}{T} \theta \left[\frac{1}{\lambda + 2(1 - \theta) + \theta} \right]$$

$$= \frac{\theta}{T} D \left[\frac{N}{N + (2 - \theta) D} \right]$$

$$= \frac{\theta}{T} D \left[1 - \frac{(2 - \theta) D}{N + (2 - \theta) D} \right] ,$$

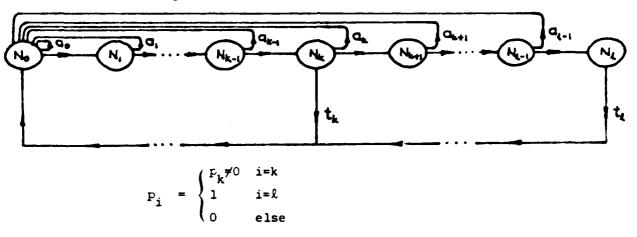
Since $2-\theta>0$, t increases monotonically with N (and approaches the limit $\frac{\theta}{T}$ D).

This fact is counter-intuitive, for one expects that if transactions behave reasonably, then the throughput should decrease when N is large because of too many restarts, even if the effect of resource contention is, ignored. We conclude, from the above result, that the memoryless assumption does not adequately model transaction behavior.

7. MULTIPLE TRANSACTION CLASSES

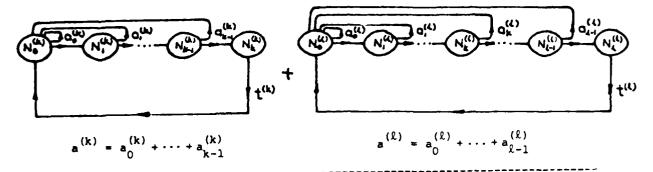
In this section, we consider a system consisting of more than one transaction class, where each class requires a different number of locks to terminate. For ease of illustration, we consider just the case of two classes, one requiring k locks, the other ℓ locks $(k < \ell)$. The model is now as in Figure 7.1.

Figure 7.1. Two Transaction Classes



If p_k is known, we can solve for p by using the polynomial f(q). Suppose, however, that we only know the ratio $b = N^{(k)}/N$, where $N = N^{(k)} + N^{(k)}$, and $N^{(k)}$, $N^{(k)}$ are the number of transactions requiring k and k locks, respectively. Then p_k is an unknown dependent on k. In this case, we may consider the system as a sum of two models, as in Figure 7.2.

Figure 7.2. Figure 7.1 as a sum of two Figure 4.1's.



Note that the probability of conflict is the same in both transaction classes. By (3.7),

$$p = \frac{1}{D} \left[\sum_{j=1}^{k} j N_{j}^{(k)} + \sum_{j=1}^{k} j N_{j}^{(k)} \right] ,$$

50

$$(1-q) = \frac{1}{D} \left[\sum_{j=1}^{k} jq^{j} N_{0}^{(k)} + \sum_{j=1}^{k} jq^{j} N_{0}^{(k)} \right] \qquad \text{by (3.9) and (4.1)}$$

$$= \frac{N}{D} \left[\frac{N_{0}^{(k)}}{N} \cdot \frac{N_{0}^{(k)}}{N_{0}^{(k)}} \cdot \sum_{j=1}^{k} jq^{j} + \frac{N_{0}^{(k)}}{N} \cdot \frac{N_{0}^{(k)}}{N_{0}^{(k)}} \cdot \sum_{j=1}^{k} jq^{j} \right]$$

$$= \lambda \begin{bmatrix} k & \ell \\ \Sigma jq^{j} & \Sigma jq^{j} \\ \frac{j=1}{k} + (1-b) \frac{j=1}{\ell} \\ \Sigma q^{j} & \Sigma q^{j} \\ j=0 & j=0 \end{bmatrix}$$
 by (3.10)

Hence q is a zero of g, where

$$-g(q) = \frac{\lambda}{(1-q)} \begin{bmatrix} k & k & k \\ \Sigma & jq^{j} & \Sigma & jq^{j} \\ b & \frac{j=1}{k} & + & (1-b) & \frac{j=1}{k} \\ \Sigma & q^{j} & \Sigma & q^{j} \\ j=0 & j=0 \end{bmatrix} - 1 \qquad (7.1)$$

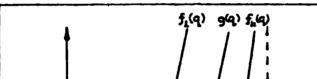
Fact 7.1: There is exactly one zero of g in [0,1).

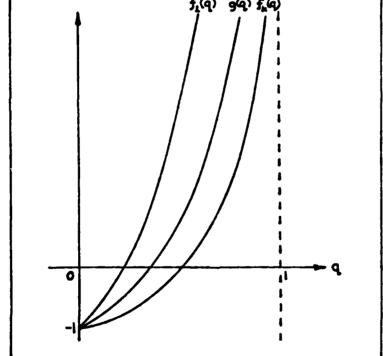
Proof. Let us make k a variable in (4.2) and index f to be f_{ν} . Then $g = bf_k + (1-b)f_{\ell}$, where--from the proof of Fact 3.2-- $f_{\alpha}(0) < 0$,

 $\lim_{\alpha} f_{\alpha}(q) = +\infty$ and f_{α} is continuous and monotonic increasing on [0,1] for $\alpha = k,\ell$. Therefore g has the same properties, to wit: g(0) < 0,

 $\lim_{x \to 0} g(q) = 0$ and g is continuous and monotonic increasing on [0,1), and so has exactly one zero in [0,1). (See Fig. 7.3).

Figure 7.3. $g = bf_k + (1-b)f_{\ell}$





As in the previous cases, this zero behaves reasonably.

Fact 7.2: Given b, p increases as λ increases, and $\lim_{n \to \infty} p = 0$, $\lim p=1.$

<u>Proof.</u> Recall from the proof of Fact 7.1 that g is continuous and monotonic increasing on [0,1). The proof that p increases as λ increases is therefore the same as that in Fact 3.3.

Let q_{α} be the zero of f_{α} in [0,1), where $\alpha=k,\ell$ and f_{α} is as in the preceding proof. Since $g=bf_k+(1-b)f_{\ell}$ where f_k and f_{ℓ} are continuous and monotonic, the zero q_{λ} in [0,1) of g must lie between q_k and q_{ℓ} . Hence p lies between p_k and p_{ℓ} , where $p_{\alpha}=1-q_{\alpha}$, $\alpha=k,\ell$. The limiting behavior of p_k and p_{ℓ} , is given by Fact 7.1, i.e. $\lim_{\lambda\to 0^+} p_{\alpha}=0$ and $\lim_{\lambda\to +\infty} p_{\alpha}=1$ for $\alpha=k,\ell$, and it follows that p has the same limits.

The variance performance measures are as follows:

From (4.4),
$$t^{(k)} = \frac{N^{(k)}}{T} \frac{(1-q)q^k}{1-q^{k+1}} = \frac{N}{T} b \frac{(1-q)q^k}{1-q^{k+1}}, \quad t^{(k)} = \frac{N}{T} (1-b) \frac{(1-q)q^k}{1-q^{k+1}}$$
(7.2)

From (4.5),
$$a^{(k)} = \frac{N}{T} b \frac{(1-q)(1-q^k)}{1-q^{k+1}}$$

$$a^{(k)} = \frac{N}{T} (1-b) \frac{(1-q)(1-q^k)}{1-q^{k+1}}$$
 (7.3)

Fact 7.3: For small λ , k and ℓ ,

(i)
$$p = \frac{\lambda}{2} [bk + (1-b)\ell]$$

(ii)
$$t = \left(\frac{b}{k+1} + \frac{1-b}{\ell+1}\right) \frac{N}{T}, \quad \text{where} \quad t = t^{(k)} + t^{(\ell)}$$

(iii)
$$a = \left(\frac{bk}{k+1} + \frac{(1-b)\ell}{\ell+1}\right) \frac{pN}{T}, \text{ where } a = a^{(k)} + a^{(\ell)}$$

<u>Proof.</u> By Fact 7.2, for small λ , k and ℓ , $q^{j} \approx 1$ for $1 \leq j \leq \ell$. Thus from (7.1),

$$0 = g(q) \approx \frac{\lambda}{p} \left[b \frac{\frac{1}{2} (k+1)k}{k+1} + (1-b) \frac{\frac{1}{2} (\ell+1)\ell}{\ell+1} \right] - 1$$
$$= \frac{\lambda}{p} \left[b \frac{k}{2} + (1-b) \frac{\ell}{2} \right] - 1$$

and so

$$p \approx \frac{\lambda}{2} [bk + (1-b)l]$$

Similarly, by (7.2),

$$t^{(k)} = \frac{N}{T} b \frac{q^k}{1+q+\dots+q^k} \approx \frac{N}{T} b \frac{1}{k+1}, \quad t^{(\ell)} \approx \frac{N}{T} (1-b) \frac{1}{\ell+1}$$

and by

$$a^{(k)} = \frac{N}{T} b \frac{1 - (1-p)^k}{1 + q + ... + q^k} \approx \frac{N}{T} b \frac{kp}{k+1}, \quad a^{(k)} \approx \frac{N}{T} (1-b) \frac{kp}{k+1}$$

Hence the claim in (i) and (ii).

Once again, our intuition about the behavior of the system when λ is small is confirmed, for we can derive the approximations in Fact 7.3 with intuitive arguments much as is done in Section 4. For example, few conflicts are expected and most transactions should complete. The average number of locks held by a length k transaction would then be k/2, and that for length ℓ transaction $\ell/2$. The average number of locks in the system would thus be

$$N^{(k)} \frac{k}{2} + N^{(k)} \frac{\ell}{2} = N \left[b \frac{k}{2} + (1-b) \frac{\ell}{2} \right], \text{ so that } p = \frac{N}{D} \left[b \frac{k}{2} + (1-b) \frac{\ell}{2} \right],$$

as i (i) above.

When considering two transaction classes, the principal interest is in how their interaction affect the performance of each. In particular, we consider now the contribution of the shorter transaction to the throughput and abort rate.

Fact 7.4: (i) $\frac{t^{(k)}}{t}$ increases as λ increases, and

$$\lim_{\lambda \to 0^+} \frac{t^{(k)}}{t} = \frac{b}{b + (1-b) \frac{k+1}{\ell+1}}, \quad \lim_{\lambda \to +\infty} \frac{t^{(k)}}{t} = 1$$

(ii) $\frac{a^{(k)}}{a}$ increases as λ increases, and

$$\lim_{\lambda \to 0^{+}} \frac{a^{(k)}}{a} = \frac{b}{b + (1-b) \frac{\ell}{\ell+1} \frac{k+1}{k}}, \quad \lim_{\lambda \to +\infty} \frac{a^{(k)}}{a} = b$$

Proof. (i) From (7.2)

$$\frac{t^{(k)}}{t} = b \frac{(1-q)q^k}{1-q^{k+1}} / \left(b \frac{(1-q)q^k}{1-q^{k+1}} + (1-b) \frac{(1-q)q^k}{1-q^{k+1}} \right)$$

$$= b / \left[b + (1-b) \frac{q^k}{1-q^{k+1}} \cdot \frac{1-q^{k+1}}{q^k} \right]$$

It is easy to show that $\frac{q^k}{1-q^{k+1}} \cdot \frac{1-q^{k+1}}{q^k}$ decreases when q decreases in [0,1), or as λ increases (Fact 7.2), so $\frac{t^{(k)}}{t}$ is an increasing function of λ .

Now for small λ , $q \approx 1$ (Fact 7.2)

$$\frac{t^{(k)}}{t} = b / \left(b + (1-b) \frac{q^{\ell}}{1+q+\ldots+q^{\ell}} \cdot \frac{1+q+\ldots+q^{k}}{q^{k}} \right)$$

$$\approx b / \left(b + (1-b) \frac{(k+1)}{(\ell+1)} \right)$$

$$= \frac{b}{b + (1-b) \frac{k+1}{\ell+1}} .$$

For large λ , $q \approx 0$, so $\frac{t^{(k)}}{t} \approx b/(b+(1-b)q^{k-k}) \approx 1$ since k > k.

(ii) From (7.3),

$$\frac{a^{(k)}}{a} = b / \left(b + (1-b) \frac{1-q^{\ell}}{1-q^{\ell+1}} \cdot \frac{1-q^{k+1}}{1-q^k} \right).$$

It is again a straightforward, if tedious, exercise to show that $\frac{a}{a}$ increases when λ increases (for $k \ge 2$).

When $\lambda \approx 0$, we have $q \approx 1$ and

$$\frac{a^{(k)}}{a} = b / \left(b + (1-b) \frac{1+q \dots + q^{\ell-1}}{1+q+\dots + q^{\ell}} \cdot \frac{1+q+\dots + q^k}{1+q+\dots + q^{k-1}} \right)$$

$$\approx b / \left(b + (1-b) \frac{\ell}{\ell+1} \cdot \frac{k+1}{k} \right) .$$

For large ,
$$q \approx 0$$
, so $\frac{a^{(k)}}{a} \approx b/(b+(1-b)) = b$.

Since transactions requiring ℓ locks continue to suffer restarts after the (k+1)-th stage, we expect $a^{(k)}/a < b$; Fact 7.4 gives

$$\frac{b}{b+(1-b)} \frac{k}{\ell+1} \frac{k+1}{k} < \frac{a^{(k)}}{a} < b$$

Furthermore, the lower bound is close to b for a wide range of values of k and l, so that the proportion of restarts is highly insensitive to the number of locks required by each class. Similarly, because more locks are required, transactions requiring l locks contribute less to the throughput. Fact 7.4_qives

$$\frac{t^{(k)}}{t} > \frac{b}{b+(1-b)\frac{k+1}{\ell+1}} > b$$

For high enough load λ , insufficient numbers make it through the extra stages, and $\frac{t^{(k)}}{t} \approx 1$.

Lessons learnt from Section 4 are also applicable here. For example, given λ and b, k and ℓ should not be greater than the lock limits for b λ and b(1- λ), respectively. Conversely, for given k and ℓ , λ and b must be chosen so that b λ and (1-b) λ are less than the saturation points of k and ℓ , respectively, or there will be no throughput to speak of for one class or the other. If λ is increased, whether the CC-thrashing point for a particular class is reached first depends on b.

8. CONCLUSIONS

We draw two sets of conclusions, one concerning what the model tells us of exclusive locking without waiting, and the other about the model itself.

If we believe that this model has captured the essential features of the locking policy studied, then it reveals the following results, assuming uniform access:

- (RI) The effect of the number of transactions on prob(conflict) is indistinguishable from that of granularity.
- (R2) For small loads, the probability of conflict, throughput and ratio of abort rate to throughput are linear in N.
- (R3) For throughput to be significant, transactions cannot request more locks than the lock limit for a given load and, conversely, the load should not exceed the saturation point for a given number of locks required. We expect the concepts of lock limit and saturation point to endure even if there is nonuniform access, when the model is extended to other locking policies, and indeed for any reasonable model for locking.
- (R4) Assuming a memoryless lock request behavior leads to increasing throughput even for large N (if resource contention is disregarded), and is unreasonable. (Hence some standard queueing models may be bad.)
- (R5) For a system with two transaction classes, the one requiring less locks contribute more to the throughput and, in fact, dominates under high loads, whilst the ratio of aborts is essentially just the ratio of transactions.

As to the desirability of the model itself, the model has the following in its favor:

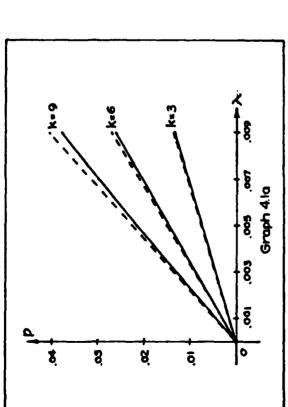
- (M1) It is simple, and makes no assumptions about probability distributions, etc.
- (M2) It cleanly separates the issues of resource contention and concurrency control, thus allowing one to study the effect of each on the system.
- (M3) It is a 'high-level' model. The model allows us to ask questions about a wide range of performance characteristics, and not just preliminary measures such as the probability of conflict. In fact, the model accepts as input the function \$\Pi\$ specifying prob(conflict) for different system states. Though it is assumed that there is uniform access in most of this paper, the assumption is not necessary for the use of the model—one can perform a numerical analysis based on a \$\Pi\$ that is empirically obtained for the particular system under study. (Restricted forms of non-uniform access can also be treated analytically [TSG].) Similarly, the model accepts any \$\Pi\$ that may be appropriate for the system concerned.
- (M4) It is flexible. It is for convenience, for instance, that T is assumed the same for different stages; if it is different, it can be handled in a way much as is done for p_i . We have also seen how the model can handle transactions of indeterminate length, and multiple transaction classes. [TSG] treats a system with queries (which share locks) and updates (which require exclusive locks).
- (M5) It has been validated for the case of all transactions requiring the same number of locks.
- (M6) It can be extended. Work is now in progress on the waiting case. It is obvious that the flow diagram in Figure 1 is equally applicable to timestamp ordering algorithms, and can be extended to more sophisticated concurrency control algorithms, such as those that abort waiting transactions.

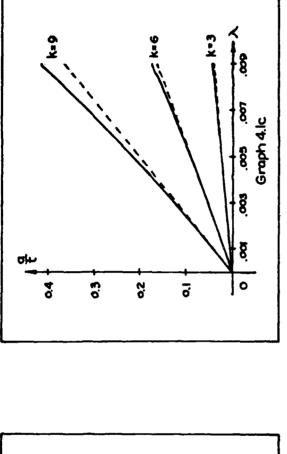
This paper contains but the first of what this model and its extensions have to offer.

Acknowledgments

We would like to thank Oded Shmueli for the use of his two-phase locking simulation program, and Ken Sevcik for his extensive and useful comments on an earlier draft of the paper.

APPENDIX 1: Graphs





---- exact solution from equations (4.2),(4.4),(4.5)
---- asymptotic solution in Fact 4.1
These graphs are plotted with D*I unit and t(N)*I unit.



₹ 680.

8

.003

8000

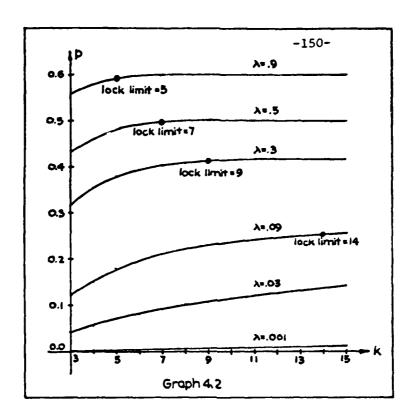
.0012

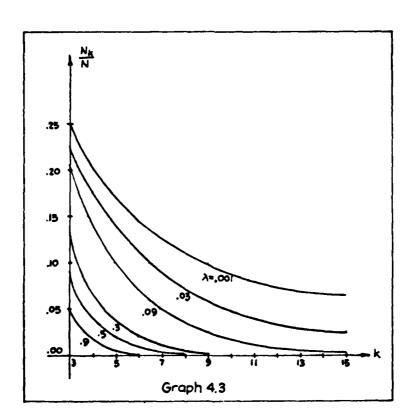
8200

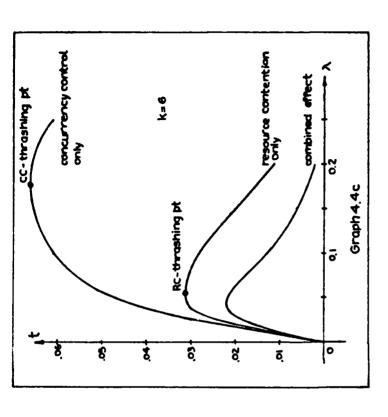
90.

.0004

Graph 4:1b



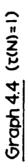




cc-thrashing pt

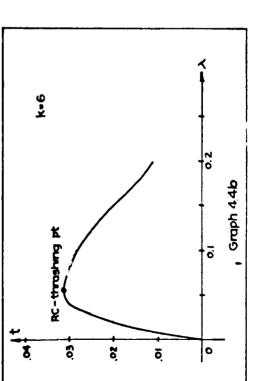
ġ

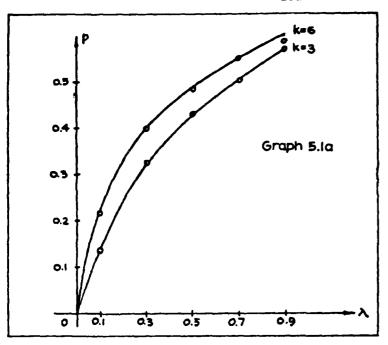
ō

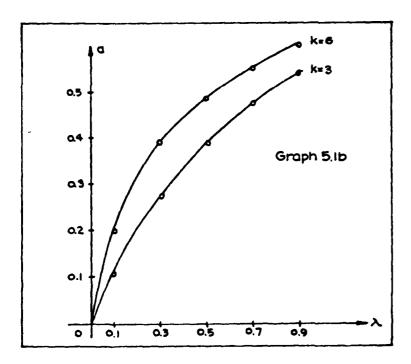


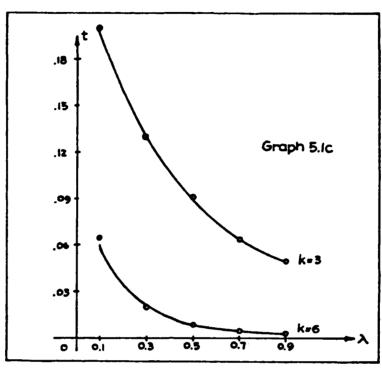
Graph 4.4a

<u>.</u>









curve obtained by MACSYMA
 value from simulation

Graph 5.1 (D=1 unit, T(N)=1 unit)

APPENDIX 2: Tables

Table 5.1

Probability of conflict p = 1-q, where q is the zero in [0,1) of f in (4.2).

Table 5.2

Unnormalized throughput $t = \frac{D}{T} \lambda \frac{(1-q)q^k}{1-q^{k+1}}$ (see (4.4)). Only the factor $\lambda \frac{(1-q)q^k}{1-q^{k+1}}$ is tabulated.

Table 5.3

Normalized throughput t' = (k+1)t; only the factor $(k+1)\lambda \frac{(1-q)q^k}{1-q^{k+1}}$ is tabulated.

Without concurrency control, the response time of a transaction is (k+1)T, and the throughput N/(k+1)T. Hence the drop in throughput in Table 5.2 when k gets larger is partly due to just the time it takes for a transaction to accumulate its locks. Table 5.3 factors this out, so as to make the effect of aborts on throughput easier to discern.

Table 5.4

Abort rate $a = \frac{D}{T} \lambda \frac{(1-q)(1-q^k)}{1-q^{k+1}}$ (see (4.5)). Only the factor $\lambda \frac{(1-q)(1-q^k)}{1-q^{k+1}}$ is tabulated.

Table 5.5

Number of aborts per transaction.

Let $u = \operatorname{prob}(a \text{ transaction completes without aborting})$. Then $u = q^k$ and expected number of aborts before completion $= \sum_{j=1}^{\infty} j(1-u)^j u = \sum_{j=1}^{\infty} (1-u)u \sum_{j=1}^{\infty} j(1-u)^{j-1} = \frac{(1-u)u}{(1-(1-u)^2)} = \frac{1-q^k}{q^k}$, which is also the ratio of abort rate to throughput (see (4.4) and (4.5)).

Table 5.6

Let $s = \frac{1}{k}$ (number of locks an average transaction holds). Number of locks in the system = Dp (see (3.7)), so $s = \frac{1}{k} \left(\frac{Dp}{N} \right) = \frac{p}{k\lambda}$.

Table 5.7

In the simulation, T=Nh, where h is the average time it takes for the scheduler to handle one request. Hence $t=\frac{1}{h} \cdot \frac{(1-q)\,q^k}{1-q^{k+1}}$ and $a=\frac{1}{h} \cdot \frac{(1-q)\,(1-q^k)}{1-q^{k+1}}$. The throughput and abort rate are given in the simulation with h as the time unit. The predicted values in this table are therefore given in the same time unit, i.e., $t=\frac{(1-q)\,q^k}{1-q^{k+1}}$ and $a=\frac{(1-q)\,(1-q^k)}{1-q^{k+1}}$.

Table 5.8

The ${\bf r}$ and ${\bf s}$ values at the CC-thrashing point for various values of ${\bf k}$.

Table 5.1

probability of conflict p

k=3									
	1	2	3	4	5	6	7	8	9
.00	0.0015	0.0030	0.0045	0.0060	0.0074	0.0089	0.0104	0.0119	0.0133
• 0	0.0148	0.0291	0.0431	0.0575	0.0706	0.0834	0.0958	0.1087	0.1205
•	0.1327	0.2343	0.3141	0.3769	0.4282	0.4706	0.5059	0.5364	0.5627
k=4									
	1	2	3	4	5	ሪ	7	3	9
.00	0.0020	0.0040	0.0060	0.0079	0.0099	0.0119	0.0138	0.0157	0.0177
• 0	0.0196	0.0385	0,0566	0.0741	0.0707	0.1071	0.1220	0.1364	0.1511
•	0.1646	0.2764	0.3561	0.4159	0.4629	0.5012	0.5334	0.5406	0.5840
k=5									
	1	2	3	4	5	6	7	3	9
.00	0.0025	0.0050	0.0074	0.0099	0.0123	0.0148	0.0171	0.0196	0.0219
.0	0.0244	0.0476	0.0689	0,0893	0.1087	0.1266	0.1438	0.1604	0.1756
•	0.1903	0.3041	0.3804	0.4366	0.4805	0.5160	0.5457	0.5710	0.5930
k=6									
	1	2	3	4	5	6	7	8	9
.00	0.0030	0.0060	0.0089	0.0119	0.0148	0,0176	0,0204	0.0232	0.0260
• 0	0.0291	0.0557	0.0800	0.1031	0.1236	0.1438	0.1618	0.1790	0.1948
•	0.2101	0.3220	0.3950	0.4481	0.4895	0.5231	0.5514	0.5757	0.5969
k=7									
	1	2	3	4	5	6	7	8	9
.00	0.0035	0.0070	0.0103	0.0137	0.0170	0.0204	0.0236	0.0269	0.0301
• 0	0.0327	0.0628	0.0901	0.1150	0.1372	0.1575	0.1763	0.1935	0.2101
•	0.2248	0.3342	0.4041	0.4547	0.4944	0.5247	0.5542	0.5779	0.5986
k=8									
	1	2	3	4	5	ሪ	7	8	9
.00	0.0040	0.0079	0.0118	0.0156	0.0193	0.0231	0.0268	0.0304	0.0338
• 0	0.0375	0.0706	0.0991	0.1251	0.1482	0.1694	0.1883	0.2057	0.2212
•	0.2361	0.3421	0.4093	0.4583	0.4970	0.5290	0.5556	0.5789	0.5954

Table 5.1 (continued)

k=9									
	1	2	3	4	5	6	7	8	9
.00	0.0045	0.0088	0.0131	0.0174	0.0216	0.0257	0.0298	0.0337	0.0376
.0	0.0412	0.0766	0.1071	0.1342	0.1575	0.1783	0.1974	0.2145	0.2302
•	0.2447	0.3473	0.4125	0.4603	0.4982	0.5296	0.5543	0.5793	0.5997
k=10									
K=10		2	-	•	5	4	7	8	9
	1	2	3	4	3	6	,	8	Y
.00	0.0050	0.0078	0.0146	0.0192	0.0238	0.0283	0.0374	0.0370	0.0412
.0	0.0453	0.0829	0.1143	0.1416	0.1653	0.1857	0.2045	0,2212	0.2366
•	0.2509	0.3511	0.4145	0,4615	0.4990	0.5301	0.5545	0.5797	0.5998
k=11									
K-11	1	2	3	4	5	6	7	8	9
	•	-	J	•			•		•
.00	0.0055	0.0108	0,0159	0,0210	0.0259	0.0307	0.0355	0.0400	0.0446
• 0	0.0494	0.0884	0.1205	0.1482	0.1715	0.1922	0.2101	0.2266	0.2418
•	0.2554	0.3532	0.4159	0.4624	0.4995	0.5303	0.5567	0.5797	0.6000
k=12									
	1	2	3	4	5	6	7	8	9
•00	0.0060	0.0117	0.0173	0.0227	0.0280	0.0332	0.0382	0.0431	0.0479
• 0	0.0521	0.0934	0.1259	0.1533	0.1763	0.1968	0.2151	0.2308	0.2459
•	0.2593	0.3548	0,4166	0.4627	0.4797	0,5305	0.5539	0.5798	0.6000
k=13									
	1	2	3	4	5	6	7	3	9
.00	0.0064	0.0125	0.0136	0.0244	0.0301	0.0355	0.0408	0.0459	0.0510
.0	0.0557	0.0975	0.1312	0.1575	0.1810	0.2006	0.2188	0.2343	0.2487
•	0.2620	0.3561	0.4167	0.4629	0.5000	0.5305	0.5569	0.5798	0.6000
k=14	1	2	3	4	5	6	7	8	9
	•	•	J	~	•	U	•	J	•
.00	0.0069	0.0135	0.0199	0.0260	0.0320	0.0377	0.0432	0.0486	0.0538
.0	0.0593	0.1015	0.1349	0.1618	0.1843	0.2038	0.2212	0.2366	0.2509
•	0.2636	0.3565	0.4172	0.4632	0.5000	0.5307	0.5569	0.5798	0.6000
k=15									
V-10	1	2	3	4	5	6	7	8	9
	•	-		-		_	-	-	
•00	0.0073	0.0144	0.0211	0.0276	0.0339	0.0398	0.0456	0.0511	0,0565
.0	0.0619	0.1047	0.1379	0.1646	0.1870	0.2063	0.2236	0.2390	0.2526
•	0.2652	0.3573	0.4176	0.4632	0.5000	0.5307	0.5569	0.5793	0.6000

Table 5.2

unnormalized throughput t (to set the real values, multiply by D/T)

k= 3									
	1	2	3	4	5	6	7	8	9
.00	0.0002	0.0005	0.0007	0.0010	0.0012	0.0015	0.0017	0.0020	0.0022
•0	0.0024	0.0048	0.0070	0.0091	0.0112	0.0131	0.0150	0.0167	0.0184
•	0.0179	0.0321	0.0390	0.0429	0.0448	0.0455	0.0454	0.0448	0.0439
k= 4									
K- 4	1	2	3	4	5	6	7	8	9
	•	-	•	7	J	· ·	,	G	7
.00	0.0002	0.0004	0.0006	0.0008	0.0010	0.0012	0.0014	0.0015	0.0017
.0	0.0019	0.0037	0.0053	0.0068	0.0082	0.0074	0.0106	0.0117	0.0126
•	0.0135	0.0189	0.0207	0.0208	0.0202	0.0192	0.0181	0.0170	0.0159
k= 5									
	1	2	3	4	5	6	7	8	9
.00	0.0002	0.0003	0.0005	0.0007	0.0008	0,0010	0.0011	0.0013	0.0014
.0	0.0016	0.0029	0.0042	0.0052	0.0061	0.0069	0.0076	0.0082	0.0088
•	0.0092	0.0112	0.0110	0.0102	0.0093	0.0083	0.0075	0.0067	0.7060
k= 6									
	1	2	3	4	5	6	7	8	9
.00	0.0001	0.0003	0.0004	0,0006	0.0007	0.0008	0.0009	0.0011	0.0012
.0	0.0013	0.0024	0.0033	0.0040	0.0046	0.0051	0.0055	0.0059	0.0061
•	0.0063	0.0067	0.0060	0.0051	0.0044	0.0037	0.0032	0.0027	0.0023
k= 7									
	1	2	3	4	5	6	7	8	9
.00	0.0001	0.0002	0.0004	0.0005	9,0006	0.0007	0.0008	0.0007	0.0010
.0	0.0011	0.0020	0.0026	0.0031	0.0035	0.0038	0.0040	0.0042	0.0043
•	0.0043	0.0040	0.0022	0.0026	0.0021	0.0017	0.0014	0.0011	0.0009
k= 8									
	1	2	3	4	5	6	7	ß	9
.00	0.0001	0.0002	0.0003	0.0004	0.0005	0.0006	0.0007	0.0008	0.0059
.0	0.0005	0.0016	0.0021	0.0025	0.0027	0.0028	0.0029	0.0030	0.0030
•	0.0030	0.0025	0.0013	0.0014	0.0010	0.0008	0.0006	0.0005	0.0004
k= 9									
, ,	1	2	3	4	5	6	7	8	9
.00	0 0001	0 0000	0 0007	0.0004	0 0005	A AA05	0.0004	0 0007	0 0000
•00	0.0001	0.0002 0.0014	0.0003	0.0004	0.0005 0.0021	0.0005 0.0021	0.0006	0.0007	0.0008
•	0.0008	0.0014	0.0017	0.0017	0.0021	0.0021	0.0021	0.0021	0.0021
•	4.444.1	4.4413	4.4410	417007	v.vvv3	V. VVV4	4.4449	V + V V V Z.	4.0001

Table 5.2 (continued)

k=10	•								
	1	2	3	4	5	6	7	8	9
.00	0.0001	0.0002	0.0003	0.0003	0.0004	0.0005	0.0005	0.0006	0.0007
. 0	0.0007	0.0011	0.0014	0.0015	0.0016	0.0016	0.0016	0.0016	0.0015
•	0.0015	0.0009	0.0006	0.0004	0.0002	0.0002	0.0001	0.0001	0.0001
k=11									
	1	2	3	4	5	6	7	8	9
.00	0.0001	0.0002	0.0002	0.0003	0.0004	0.0004	0.0005	0.0005	0.0006
.0	0.0006	0.0010	0.0011	0.0012	0.0012	0.0012	0.0012	0.0011	0.0011
•	0.0010	0.0006	0.0003	0.0002	0.0001	0.0001	0.0001	0.0000	0.0000
k=12									
	1	2	3	4	5	6	7	8	9
.00	0.0001	0.0001	0.0002	0,0003	0,0003	0.0004	0.0004	0.0005	0.0005
.0	0.0005	0.0008	0,0009	0.0009	0.0009	0.0009	0.0009	3000.0	0.0008
•	0.0007	0.0004	0,0002	0.0001	0.0001	0.0000	0.0000	0.0000	0.0000
k=13									
	1	2	3	4	5	6	7	8	9
.00	0.0001	0.0001	0.0002	0.0002	0.0003	0.0003	0,0004	0.0004	0.0004
.0	0.0005	0.0007	0.0007	0.0007	0.0007	0.0007	0.0006	3000.0	0.0006
•	0.0005	0.0002	0.0001	0.0001	0.0000	0.0000	0.0000	0.0000	0.0000
k=14									
	1	2	3	4	5	6	7	8	9
.00	0.0001	0.0001	0.0002	0.0002	0.0003	0.0003	0.0003	0,0004	0.0004
• 0	0.0004	0.0006	0.0006	0.0006	0.0006	0.0005	0.0005	0.0004	0.0004
•	0.0004	0.0001	0.0001	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000
k=15									
	1	2	3	4	5	6	7	8	9
.00	0.0001	0.0001	0.0002	0.0002	0,0002	0.0003	0.0003	0.0003	0.0004
.0	0.0004	0.0005	0.0005	0.0005	0.0004	0.0004	0.0004	0.0003	0.0003
•	0.0003	0.0001	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000

Table 5.3

normalized throughput t'=(k+1)\$t (to set the real values; multiply by D/T)

k= 3									
	1	2	3	4	5	6	7	8	9
.00	0.0010	0.0020	0.0030	0.0040	0.0049	0.0059	0.0069	0.0077	0.0088
.0	0.0098	0.0191	0.0280	0.0365	0.0446	0.0524	0.0598	0.0668	0.0735
•	0.0798	0.1282	0.1562	0.1718	0.1792	0.1818	0.1817	0.1793	0,1758
k= 4									
	1	2	3	4	5	6	7	8	9
.00	0.0010	0.0020	0.0030	0,0039	0.0049	0,0059	0.0068	0.0077	0.0087
.0	0.0096	0.0185	0.0266	0.0341	0.0409	0.0472	0.0531	0.0584	0.0631
•	0.0676	0.0745	0.1033	0.1039	0.1008	0.0930	0.0905	0.0350	0.0797
k= 5									
	1	2	3	4	5	6	7	8	9
.00	0.0010	0.0020	0.0027	0.0039	0.0048	0.0058	0,0067	0.0076	0.0085
• 0	0.0094	0.0176	0.0249	0.0313	0.0368	0.0416	0.0458	0.0494	0.0526
•	0.0553	0.0372	0.0663	0.0614	0.0556	0.0500	0.0448	0.0401	0.0359
k= 6									
	1	2	3	4	5	6	7	8	9
.00	0,0010	0.0020	0.0029	0.0039	0,0048	0,0057	0.0066	0.0074	0.0083
• 0	0,0071	0.0167	0.0230	0.0282	0,0325	0.0359	0.0388	0.0410	0.0428
•	0.0442	0.0469	0.0419	0.0360	0.0306	0.0260	0.0221	0.0187	0.0162
k= 7									
	1	2	3	4	5	6	7	8	9
.00	0.0010	0.0020	0.0029	0.0038	0.0047	0.0056	0.0054	0.0073	0.0081
•,0	0.0089	0.0158	0.0211	0.0251	0,0282	0.0305	0.0322	0.0335	0.0342
•	0.0348	0.0323	0.0243	0,0210	0.0168	0.0135	0.0109	0.0088	0.0072
k= 8									
	1	2	3	4	5	6	7	8	9
.00	0.0010	0.0019	0.0029	0.0038	0.0046	0.0055	0.0063	0.0070	0.0078
• 0	0.0085	0.0147	0.0171	0.0221	0.0242	0.0255	0.0264	0.0268	0.0271
•	0.0270	0,0221	0.0165	0.0123	0.0092	0.0069	0.0053	0.0041	0.0032

Table 5.3 (continued)

k= 9	•								
•	1	2	3	4	5	ሪ	7	8	9
		0 0010	0.0000		A AAAE	0 00ET	0.00/1	0.0068	A AA7E
•00	0.0010	0.0019 0.0136	0.0028	0.0037	0.0045	0.0053	0.0061	0.0088	0,0075 0,0212
•0	0.0062	0,0136	0.0171	0.0172	0.0203	0.0213	0.0059	0.0015	0.0014
•	0.0208	0,0132	V. O. 1.04	V. VV/2	V • V V S V	V+VV30	4.44.0	V+VV1/	0.0014
k=10									
	1	2	3	4	5	6	7	8	9
.00	0,0010	0.0019	0.0028	0.0036	0.0044	0.0052	0.0057	0.0066	0.0072
•0	0.0078	0.0125	0.0152	0.0166	0.0173	0.0175	0.0174	0.0171	0.0166
•	0.0160	0.0103	0.0065	0,0042	0.0027	0.0018	0.0013	0.0009	0.0006
k=11									
¥-11	•	2	7	•	-	4	7		9
	1	2	3	4	5	6	7	8	7
.00	0.0010	0.0019	0.0027	0.0036	0.0043	0.0050	0.0057	0.0063	0.0069
.0	0.0075	0.0114	0.0134	0.0143	0.0145	0.0143	0.0140	0.0135	0.0129
•	0.0123	0.0071	0.0040	0.0024	0,0015	0.0009	0.0003	0.0004	0.0003
k=12									
W-12	1	2	3	4	5	6	7	8	9
	•	-	•	•	•	•	•		•
.00	0.0010	0.0019	0.0027	0,0035	0,0042	0.0049	0.0055	0.0061	0.0066
• 0	0.0071	0.0104	0.0118	0.0122	0.0122	0.0117	0.0112	0.0107	0.0100
•	0.0094	0.0048	0.0025	0,0014	0.0008	0.0005	0.0003	0.0002	0.0001
k=13									
W-13	1	2	3	4	5	6	7	8	9
	•	-	•	•	•		·	_	•
.00	0.0010	0.0018	0.0026	0.0034	0.0041	0.0047	0.0053	0.0058	0.0063
• C	0.0067	0.0094	0.0103	0.0104	0.0101	0.0096	0.0089	0.0084	0.0078
•	0.0072	0.0033	0.0016	0,0008	0,0004	0.0002	0.0001	0.0001	0.0001
k=14									
W-24	1	2	3	4	5	6	7	3	9
	_	_	_	·	_				
.00	0.0010	0.0018	0,0026	0,0033	0,0039	0,0045	0.0050	0.0055	0.0059
.0	0.0063	0.0085	0,0090	9800.0	0.0084	0.0078	0.0072	0.0066	0,0060
•	0.0055	0.0022	0.0010	0.0005	0.0002	0.0001	0.0001	0.0000	0.0000
k=15									
- 	1	- 2	3	4	5	6	7	8	9
	•	-	•		•	_	•	_	•
.00	0.0009	0.0018	0.0025	0.0032	0.0038	0.0043	0.0048	0.0052	0.0056
• 0	0.0059	0.0077	0.0079	0.0075	0.0070	E 600,0	0.0057	0.0052	0.0047
•	0.0042	0.0015	0.9906	0.0003	0.0001	0.0001	0.0000	0.0000	0.0000

Table 5.4

abort rate a (to set the real values, multiply by D/T)

k= 3									
	1	2	3	4	5	6	7	8	9
.00	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0001	0.0001	0.0001
•0	0.0001	0.0004	0.0010	0.0018	0.0027	0.0039	0,0053	0,0069	0.0086
•	0.0106	0.0373	0.0820	0.1346	0.1949	0.2610	0.3312	0.4051	0.4817
k= 4					_				
	1	2	3	4	5	6	7	8	9
.00	0.0000	0.0000	0,0000	0.0000	0,0000	0.0001	0.0001	0.0001	0.0001
•0	0.0002	0.0000	0.0014	0.0025	0.0038	0.0054	0.0072	0.0093	0.0001
• •	0.0142	0.0501	0.0975	0.1577	0.2221	0.2911	0.3637	0.4390	0.5163
•		0,,,,,,,							
k= 5									
	1	2	3	4	5	6	7	8	9
							• • • • •		
.00	0,0000	0,0000	0,0000	0,0000	0.0001	0.0001	0.0001	0.0001	0.0002
• 0	0.0002	0.0008	0.0018	0.0031	0.0048	0.0067	0.0090	0.0115	0.0143
•	0.0173	0.0574	0.1079	0.1702	0.2358	0.3053	0.3779	0.4530	0.5301
k= 6									
••	1	2	3	4	5	6	7	8	9
.00	0.0000	0.0000	0.0000	0.0000	0.0001	0.0001	0.0001	0.0003	0.0002
• 0	0.0003	0.0010	0.0021	0.0037	0.0056	0.0079	0.0104	0.0133	0.0163
•	0.0197	0.0623	0.1161	0.1789	0.2426	0.3119	0.3842	0.4590	0.5359
k= 7									
K- /	1	2	3	4	5	6	7	8	9
	-	•	Ü	-	U	U	•	· ·	•
.00	0.0000	0.0000	0.0000	0.0000	0.0001	0.0001	0.0001	0.0002	0.0002
.0	0.0003	0.0011	0.0025	0.0042	0.0064	0.0089	0.0116	0.0147	0.0180
•	0.0215	0.0655	0.1199	0.1807	0,2462	0.3152	0.3872	0.4617	0.5382
k= 8			_		_		_	_	_
	1	2	3	4	5	6	7	8	9
.00	0.0000	0.0000	0.0000	0.0001	0.0001	0.0001	0.0002	0.0002	0.0003
.0	0.0003	0.0013	0.0028	0.0047	0.0070	0.0077	0.0126	0.0158	0,0192
•	0.0229	0.0676	0.1220	0.1827	0.2480	0.3170	0,3886	0.4627	0.5392
					-				

Table 5.4 (continued)

k= 9	
1 2 3 4 5 6 7	E 9
.00 0.0000 0.0000 0.0000 0.0001 0.0001 0.0001 0.0002 0.	.0002 0.0003
.0 0.0004 0.0014 0.0030 0.0051 0.0076 0.0103 0.0134 0.	.0167 0.0202
. 0.0240 0.0489 0.1233 0.1838 0.2489 0.3176 0.3893 0.	.4633 0.5396
k=10	
1 2 3 4 5 6 7	8 9
.00 0.0000 0.0000 0.0000 0.0001 0.0001 0.0002 0.0002 0.	.0003 0.0003
	.0174 0.0209
. 0.0247 0.0699 0.1241 0.1344 0.2494 0.3180 0.3895 0	,4637 0.5398
k=11	
1 2 3 4 5 6 7	8 9
.00 0.0000 0.0000 0.0000 0.0001 0.0001 0.0002 0.0002 0.	.0003 0.0004
	0179 0.0215
	.4637 0.5400
k=12	
1 2 3 4 5 6 7	8 9
.00 0,0000 0.0000 0.0000 0.0001 0.0001 0.0002 0.0003 0.	.0003 0.0004
	.0183 0.0219
. 0.0257 0.0708 0.1247 0.1850 0.2498 0.3183 0.3893 0.	.4639 0.5400
k=13	_
1 2 3 4 5 6 7	8 9
.00 0.0000 0.0000 0.0001 0.0001 0.0001 0.0002 0.0003 0.	.0003 0.0004
.0 0.0005 0.0019 0.0038 0.0062 0.0089 0.0119 0.0152 0.	0186 0.0222
	.4337 0.5400
k=14	
1 2 3 4 5 6 7	8 9
	-
.00 0.0000 0.0000 0.0001 0.0001 0.0002 0.0002 0.0003 0.	.0004 0.0005
	.0188 0.0225
	.4639 0.5400
L-18	
k=15 1 2 3 4 5 6 7	8 9
1 2 3 4 5 6 7	7
.00 0.0000 0.0000 0.0001 0.0001 0.0002 0.0002 0.0003 0.	.0004 0.0005
	0190 0.0227
	.4639 0.5400

the ratio r of abort rate to throughput also the number of aborts per transaction

k= 3									
	1	2	3	4	5	6	7	8	9
.00	0.0045	0,0090	0.0136	0.0181	0.0227	0.0272	0.0318	0.0364	0.0410
.0	0.0457	0.0927	0.1412	0.1944	0.2458	0.2786	0.3529	0.4125	0.4679
•	0.5328	1.2276	2,0974	3.1345	4.3502	5.7406	7.2915	9.0358	10.961
•	010020	1121/0	2.70774	0,720,70	,,,,,,,			, , , , , , , ,	
k= 4									
	1	2	3	4	5	6	7	8	9
.00	0.0080	0.0161	0.0242	0.0324	0.0406	0.0489	0.0572	0.0656	0.0740
.0	0.0824	0.1699	0.2625	0.3605	0.4641	0.5735	0.4830	0.7982	0.9257
•	1.0529	2.4478	4.8168	7.5904	11,020	15.160	20.070	25.834	32.399
k= 5		_	_		-	,	-	0	9
	1	2	3	4	5	6	7	8	7
.00	0.0126	0.0253	0,0381	0.0510	0,0641	0,0773	0.0901	0.1041	9.1171
• 0	0.1314	0.2763	0.4290	0.5959	0.7781	0.9680	1.1738	1.3964	1.6261
•	1,8730	5.1275	9.9526	16.619	25.433	36.640	50.653	67.819	88.541
k= 6		•	_	4	_	,	-	•	9
	1	2	3	4	5	6	7	8	7
.00	0.0181	0.0365	0.0552	0.0742	0.0934	0.1123	0.1315	0.1516	0.1713
.0	0.1941	0.4105	0.6496	0.9215	1,2066	1,5390	1.8830	2.2650	2,6705
•	3,1172	9.2980	19,400	34.395	55.520	34.033	121.64	170,45	232.21
k= 7	•	2	3	4	5	6	7	8	9
	1	Æ.	3	7	3	6	,	U	•
.00	0.0248	0.0500	0,0751	0.1015	0 - 1276	0,1550	0.1822	0.2100	0.2383
.0	0.2637	0.5745	0.9363	1.3526	1.8092	2.3201	2.8863	3,5077	4.2124
•	4.9447	16.246	36,457	68.790	117.46	187.06	284.63	417.75	594.13
k= 8									
*- 0	1	2	3	4	5	6	7	8	9
	A	∠ .	J.	7	J	J	•		•
.00	0.0325	0.49658	0.0773	0.1336	0.1689	0.2052	0.2424	0.2798	0.3148
.0	0.3581	0.7968	1.3045	1.9132	2,6087	3,4158	4.3074	5,3126	6,3879
•	7,6202	27,493	66.492	133,85	242.96	411.66	655.84	1011.2	1505.4

Table 5.5 (continued)

1	k= 9	•								
.0	•		2	3	4	5	6	7	8	9
	-00	0.0412	0.0830	0.1263	0.1711	0,2174	0.2643	0.3128	0.3617	0.4123
k=10 1 2 3 4 5 6 7 8 9 .00 0.0511 0.1035 0.1583 0.2142 0.2726 0.3322 0.4633 0.4575 0.5235 .0 0.5905 1.3761 2.3646 3.6053 5.0893 6.7980 6.8481 11.180 13.883 .0 0.5905 1.3761 2.3646 3.6053 5.0893 6.7980 6.8481 11.180 13.883 .0 0.0622 0.1267 0.1934 0.2623 0.3348 0.4096 0.4881 0.5674 0.6521 .0 0.7465 1.7687 3.1055 4.8391 6.9210 9.4695 12.389 15.888 20.022 .2 24.635 119.58 369.28 920.81 2024.5 4072.1 7703.2 13813 23840 k=12 1 2 3 4 5 6 7 8 9 .00 0.0744 0.1512 0										
1	•									
.00	k=10									
. 0 0 0.5905 1.3761 2.3646 3.6053 5.0893 6.7980 8.8481 11.180 13.883		1	2	3	4	5	6	7	8	9
. 0 0 0.5905 1.3761 2.3646 3.6053 5.0893 6.7980 8.8481 11.180 13.883	.00	0.0511	0.1035	0.1583	0.2142	0.2726	0.3322	0.4633	0.4573	0.5235
k=11 1 2 3 4 5 6 7 8 9 .00 0.0622 0.1267 1.7687 2.1055 3.1055 4.8391 0.7465 1.7687 2.1055 4.8391 0.7465 1.7687 2.1055 4.8391 0.74075 12.388 15.888 20.022 2.4635 119.58 20.022 2.2467 2.0652 10.022 2.00222 2.00222 2.0022 2.00222 2.00222 2.00222 2.00222 2.00222 2.00222 2.00222 2.00222 2.00222										
1 2 3 4 5 6 7 8 9 .00 0.0622 0.1267 0.1934 0.2623 0.3348 0.4076 0.4881 0.5674 0.6521 .0 0.7465 1.7687 3.1055 4.8391 6.9210 9.4695 12.389 15.888 20.022 .0 24.635 119.58 369.28 920.81 2024.5 4072.1 7703.2 13813 23840 k=12 1 2 3 4 5 6 7 8 9 .00 0.0744 0.1512 0.2327 0.3168 0.4060 0.4989 0.5955 0.6959 0.6959 22.298 28.547 .0 0.9012 2.2427 4.0247 6.3621 9.2476 12.868 17.282 22.298 28.547 .0 1.1069 2.7333 3.2227 0.3785 0.4372 0.5997 0.7179 0.6418 0.9739 .0 1.1069 2.7933 5.2227 8.2867 12.406 17.380 23.758 31.155 40.	•	16.981	74,513	210.23	486.66		1903.2	3398.8	5805.9	9497.6
1 2 3 4 5 6 7 8 9 .00 0.0622 0.1267 0.1934 0.2623 0.3348 0.4076 0.4881 0.5674 0.6521 .0 0.7465 1.7687 3.1055 4.8391 6.9210 9.4695 12.389 15.888 20.022 .0 24.635 119.58 369.28 920.81 2024.5 4072.1 7703.2 13813 23840 k=12 1 2 3 4 5 6 7 8 9 .00 0.0744 0.1512 0.2327 0.3168 0.4060 0.4989 0.5955 0.6959 0.6959 22.298 28.547 .0 0.9012 2.2427 4.0247 6.3621 9.2476 12.868 17.282 22.298 28.547 .0 1.1069 2.7333 3.2227 0.3785 0.4372 0.5997 0.7179 0.6418 0.9739 .0 1.1069 2.7933 5.2227 8.2867 12.406 17.380 23.758 31.155 40.	k=11									
.0	**	1	2	3	4	5	6	7	8	9
.0	.00	0,0622	0.1267	0,1934	0.2323	0.3348	0.4096	0.4381	0.5674	0.6521
k=12 1 2 3 4 5 6 7 8 9 .00 0.0744 0.1512 0.232? 0.3168 0.4060 0.4989 0.5955 0.6959 0.8020 . 0.7012 2.2427 4.0247 6.3621 7.2476 12.868 17.282 22.298 28.547 . 5.6442 191.30 641.07 1724.6 4070.4 8719.7 17472 33030 59603 k=13 1 2 3 4 5 6 7 8 9 .00 0.0865 0.1783 0.2772 0.3785 0.4872 0.5997 0.7179 0.8418 0.9739 .0 1.1069 2.7933 5.2227 8.2867 12.406 17.380 23.758 31.155 40.144 * 50.905 304.65 1107.2 3232.9 8191.0 18574 39436 78613 149010 * 4 5 6 7 8 9 .00 0.1011 0.2099 0.3249 0.446	• 0	0.7465	1.7687	3,1055	4.8391		9.4695	12,385	15.888	20.022
1 2 3 4 5 6 7 8 9 .00 0.0744 0.1512 0.2327 0.3168 0.4060 0.4989 0.5955 0.6959 0.8020 .0 0.7012 2.2427 4.0247 6.3621 7.2476 12.868 17.282 22.228 28.547 . 5.6442 191.30 641.87 1724.6 4070.4 8719.7 17472 33030 59603 k=13 1 2 3 4 5 6 7 8 9 .00 0.0865 0.1783 0.2772 0.3785 0.4872 0.5997 0.7179 0.8418 0.9739 .0 1.1069 2.7933 5.2227 8.2867 12.406 17.380 23.758 31.155 40.144 . 50.905 304.65 1107.2 3232.9 8191.0 18574 39436 78613 149010 k=14 1 2 3 4 5 6 7 8 9 .00 0.1011 0.2099 0.3249 0.4461 0.5776 0.7131 0.8569 1.0092 1.1701 .0 1.3522 3.4764 6.6104 10.829 16.332 23.313 32.106 42.832 56.113 . 71.543 477.97 1918.7 6066.0 16383 39825 89009 167100 372528 k=15 1 2 3 4 5 6 7 8 9 .00 0.1169 0.2429 0.3777 0.5221 0.6778 0.8403 1.0145 1.1978 1.3932 .0 0.1169 0.2429 0.3777 0.5221 0.6778 0.8403 1.0145 1.1978 1.3932 .0 0.1169 0.2429 0.3777 8.525	•	24,635	119.58	369.28	920.31	2024.5	4072.1	7703.2	13813	23840
.00	k=12									
. 0 0.9012 2.2427 4.0247 6.3621 9.2476 12.868 17.282 22.298 28.547 5.6442 191.30 641.37 1724.6 4070.4 8719.7 17472 33030 59603 k=13 1 2 3 4 5 6 7 8 9 .00 0.0865 0.1783 0.2772 0.3785 0.4872 0.5997 0.7179 0.8418 0.9739 .0 1.1069 2.7933 5.2227 8.2867 12.406 17.380 23.758 31.155 40.144 . 50.905 304.65 1107.2 3232.9 8191.0 18574 39436 78613 149010 k=14 1 2 3 4 5 6 7 8 9 .00 0.1011 0.2099 0.3249 0.4461 0.5776 0.7131 0.8569 1.0092 1.1701 .0 1.3522 3.4764 6.6104 10.829 16.332 23.313 32.106 42.832 56.113 . 71.543 477.97 1918.7 6066.0 16383 37825 89009 167100 372528 k=15 1 2 3 4 5 6 7 8 9 .00 0.1169 0.2429 0.3777 0.5221 0.6778 0.8403 1.0145 1.1978 1.3932 .0 1.6083 4.2577 8.2655 13.839 21.314 31.030 43.538 59.107 77.857		1	2	3	4	5	6	7	8	9
k=13 1 2 3 4 5 6 7 8 9 .00 0.0865 0.1783 0.2772 0.3785 0.4872 0.5997 0.7179 0.8418 0.9739 .0 1.1069 2.7933 5.2227 8.2867 12.406 17.380 23.758 31.155 40.144 .0 50.905 304.65 1109.2 3232.9 8191.0 18574 39436 76613 149010 k=14 1 2 3 4 5 6 7 8 9 .00 0.1011 0.2099 0.3249 0.4461 0.5776 0.7131 0.8569 1.0092 1.1701 .0 1.3522 3.4764 6.6104 10.829 16.322 23.313 32.106 42.832 56.113 .71.543 477.97 1918.7 6066.0 16383 39825 89009 167100 372528 .00 0.1169 0.2429 0.3777 0.5221 0.6778 0.8403 1.0145 1.1978 1.3932 .00 <td>.00</td> <td>0.0744</td> <td>0.1512</td> <td>0,2329</td> <td>0.3168</td> <td>0.4060</td> <td>0.4989</td> <td>0.5955</td> <td>0.6959</td> <td>0.8020</td>	.00	0.0744	0.1512	0,2329	0.3168	0.4060	0.4989	0.5955	0.6959	0.8020
k=13 1 2 3 4 5 6 7 8 9 .00 0.0865 (0.1783) (0.1783) (0.2772) (0.3785) (0.4372) (0.5977) (0.5977) (0.7179) (0.8418) (0.9739) (0.1069) (0.1069) (0.9739) (0.905)	• 0	0.9012	2,2427	4.0247	6.3621	9.2476	12.868	17,282	22,298	28.547
1 2 3 4 5 6 7 8 9 .00 0.0865 0.1783 5.2227 8.2867 12.406 17.380 23.758 31.155 40.144 . 50.905 304.65 1107.2 3232.9 8191.0 18574 39436 78613 149010 k=14 1 2 3 4 5 6 7 8 9 .00 0.1011 0.2099 0.3249 0.4461 0.5776 0.7131 0.8569 1.0092 1.1701 .0 1.3522 3.4764 6.6104 10.829 16.332 23.313 32.106 42.832 56.113 . 71.543 477.97 1918.7 6066.0 16383 39825 89009 167100 372526 k=15 1 2 3 4 5 6 7 8 9 .00 0.1169 0.2429 0.3777 0.5221 0.6778 0.8403 1.0145 1.1978 1.3932 .0 0.1169 0.2429 0.3777 0.5221 0.6778 0.8403 1.0145 59.107 77.857	•	5.6442	191.30	641.37	1724.6	4070.4	8719.7	17472	33030	59603
1 2 3 4 5 6 7 8 9 .00 0.0865 0.1783 5.2227 8.2867 12.406 17.380 23.758 31.155 40.144 . 50.905 304.65 1107.2 3232.9 8191.0 18574 39436 78613 149010 k=14 1 2 3 4 5 6 7 8 9 .00 0.1011 0.2099 0.3249 0.4461 0.5776 0.7131 0.8569 1.0092 1.1701 .0 1.3522 3.4764 6.6104 10.829 16.332 23.313 32.106 42.832 56.113 . 71.543 477.97 1918.7 6066.0 16383 39825 89009 167100 372526 k=15 1 2 3 4 5 6 7 8 9 .00 0.1169 0.2429 0.3777 0.5221 0.6778 0.8403 1.0145 1.1978 1.3932 .0 0.1169 0.2429 0.3777 0.5221 0.6778 0.8403 1.0145 59.107 77.857	k=13									
.0 1.1069 2.7933 5.2227 8.2867 12.406 17.380 23.758 31.155 40.144 .0 50.905 304.65 1109.2 3232.9 8191.0 18574 39436 78613 149010 k=14 1 2 3 4 5 6 7 8 9 .00 0.1011 0.2099 0.3249 0.4461 0.5776 0.7131 0.8569 1.0092 1.1701 .0 1.3522 3.4764 6.6104 10.829 16.332 23.313 32.106 42.832 56.113 .71.543 477.97 1918.7 6066.0 16383 39825 89009 187100 372528 k=15 1 2 3 4 5 6 7 8 9 .00 0.1169 0.2429 0.3779 0.5221 0.6778 0.8403 1.0145 1.1978 1.3932 .0 1.6083 4.2577 8.2655 13.839 21.314 31.030 43.538 59.107 77.857		1	2	3	4	5	6	7	8	9
.0 1.1069 2.7933 5.2227 8.2867 12.406 17.380 23.758 31.155 40.144 .0 50.905 304.65 1109.2 3232.9 8191.0 18574 39436 78613 149010 k=14 1 2 3 4 5 6 7 8 9 .00 0.1011 0.2099 0.3249 0.4461 0.5776 0.7131 0.8569 1.0092 1.1701 .0 1.3522 3.4764 6.6104 10.829 16.332 23.313 32.106 42.832 56.113 .71.543 477.97 1918.7 6066.0 16383 39825 89009 187100 372528 k=15 1 2 3 4 5 6 7 8 9 .00 0.1169 0.2429 0.3779 0.5221 0.6778 0.8403 1.0145 1.1978 1.3932 .0 1.6083 4.2577 8.2655 13.839 21.314 31.030 43.538 59.107 77.857	•00	0.0865	0.1783	0.2772	0.3785	0.4372	0.5997	0.7179	0.8418	0.9739
. 50,905 304.65 1107.2 3232.9 8191.0 18574 39436 78613 149010 k=14 1 2 3 4 5 6 7 8 9 .00 0.1011 0.2099 0.3249 0.4461 0.5776 0.7131 0.8569 1.0092 1.1701 .0 1.3522 3.4764 6.6104 10.829 16.332 23.313 32.106 42.832 56.113 . 71.543 477.97 1918.7 6066.0 16383 39825 89009 187100 372528 k=15 1 2 3 4 5 6 7 8 9 .00 0.1169 0.2429 0.3779 0.5221 0.6778 0.8403 1.0145 1.1978 1.3932 .0 1.6083 4.2577 8.2655 13.839 21.314 31.030 43.538 59.107 77.857	.0								31,155	40.144
1 2 3 4 5 6 7 8 9 .00 0.1011 0.2099 0.3249 0.4461 0.5776 0.7131 0.8569 1.0092 1.1701 .0 1.3522 3.4764 6.6104 10.829 16.332 23.313 32.106 42.832 56.113 . 71.543 477.97 1918.7 6066.0 16383 39825 89009 187100 372528 k=15 1 2 3 4 5 6 7 8 9 .00 0.1169 0.2429 0.3779 0.5221 0.6778 0.8403 1.0145 1.1978 1.3932 .0 1.6083 4.2577 8.2655 13.839 21.314 31.030 43.538 59.107 77.857	•	50,905				8191.0		39436	78613	149010
1 2 3 4 5 6 7 8 9 .00 0.1011 0.2099 0.3249 0.4461 0.5776 0.7131 0.8569 1.0092 1.1701 .0 1.3522 3.4764 6.6104 10.829 16.332 23.313 32.106 42.832 56.113 . 71.543 477.97 1918.7 6066.0 16383 39825 89009 187100 372528 k=15 1 2 3 4 5 6 7 8 9 .00 0.1169 0.2429 0.3779 0.5221 0.6778 0.8403 1.0145 1.1978 1.3932 .0 1.6083 4.2577 8.2655 13.839 21.314 31.030 43.538 59.107 77.857	k=14									
.0 1.3522 3.4764 6.6104 10.829 16.332 23.313 32.106 42.832 56.113 .71.543 477.97 1918.7 6066.0 16383 39825 89009 167100 372528 k=15 1 2 3 4 5 6 7 8 9 .00 0.1169 0.2429 0.3777 0.5221 0.6778 0.8403 1.0145 1.1978 1.3932 .0 1.6083 4.2577 8.2655 13.839 21.314 31.030 43.538 59.107 77.857		1	2	3	4	5	6	7	8	9
.0 1.3522 3.4764 6.6104 10.829 16.332 23.313 32.106 42.832 56.113 .71.543 477.97 1918.7 6066.0 16383 39825 89009 167100 372528 k=15 1 2 3 4 5 6 7 8 9 .00 0.1169 0.2429 0.3777 0.5221 0.6778 0.8403 1.0145 1.1978 1.3932 .0 1.6083 4.2577 8.2655 13.839 21.314 31.030 43.538 59.107 77.857	.00	0.1011	0.2099	0.3249	0.4461	0,5776	0.7131	0.8569	1.0092	1.1701
. 71,543 477.97 1918.7 6066.0 16383 39825 89009 187100 372528 K=15 1 2 3 4 5 6 7 8 9 .00 0.1169 0.2429 0.3777 0.5221 0.6778 0.8403 1.0145 1.1978 1.3932 .0 1.6083 4.2577 8.2655 13.839 21.314 31.030 43.538 59.107 77.857	.0	1.3522			10.829	16,332	23,313	32.106	42.832	56.113
1 2 3 4 5 6 7 8 9 -00 0.1169 0.2429 0.3779 0.5221 0.6778 0.8403 1.0145 1.1978 1.3932 -0 1.6083 4.2577 8.2655 13.839 21.314 31.030 43.538 59.107 77.857								89009	187100	372528
1 2 3 4 5 6 7 8 9 -00 0.1169 0.2429 0.3779 0.5221 0.6778 0.8403 1.0145 1.1978 1.3932 -0 1.6083 4.2577 8.2655 13.839 21.314 31.030 43.538 59.107 77.857	k=15									
.0 1.6083 4.2577 8.2655 13.839 21.314 31.030 43.538 59.107 77.857		1	2	3	4	5	6	7	8	9
.0 1.6083 4.2577 8.2655 13.839 21.314 31.030 43.538 59.107 77.857	-00	0.1169	0.2429	0.3779	0.5221	0.4779	0.8403	1.0145	1.1978	1.3932
	_									

Table 5.6

s = (number of locks an average transaction holds)/k

k = 3									
	1	2	3	4	5	6	7	8	9
.00	0.4993	0.4985	0.4978	0.4770	0.4963	0.4955	0.4948	0.4941	0.4933
.0	0.4926	0.4854	0.4785	0.4791	0.4709	0.4634	0.4564	0.4531	0.4463
•	0.4423	0.3905	0,3490	0.3141	0.2855	0.2315	0.2409	0.2235	0.2084
k= 4									
	1	2	3	4	5	6	7	8	9
.00	0.4990	0.4980	0.4970	0.4960	0.4950	0.4941	0.4931	0.4921	0.4912
.0	0.4902	0.4808	0.4717	0.4630	0.4545	0.4464	0.4358	0.4264	0.4197
•	0.4114	0.3455	0.2967	0.2579	0.2315	0.2089	0.1905	0.1752	0.1622
k= 5									
	1	2	3	4	5	6	7	8	9
.00	0.4988	0.4975	0.4963	0.4950	0,4938	0.4926	0.4886	0.4902	0.4869
.0	0.4878	0.4762	0.4593	0.4463	0.4349	0.4221	0.4110	0.4009	0.3902
•	0.3804	0.3041	0.2536	0.2183	0,1922	0.1720	0.1559	0.1427	0.1318
k= 6									
	1	2	3	4	5	6	7	8	9
.00	0.4985	0.4970	0,4955	0.4741	0.4926	0.4885	0.4851	0.4843	0.4816
.0	0.4854	0.4643	0.4446	0.4297	0.4119	0.3995	0.3852	0:3729	8035,0
•	0.3502	0.2684	0.2195	0.1867	0.1632	0.1453	0.1313	0.1199	0.1105
k= 7									
	1	2	3	4	5	6	7	8	9
.00	0.4983	0.4965	0.4701	0,4896	0.4859	0.4851	0.4822	0.4795	0.4773
.0	0.4697	0.4485	0.4290	0.4109	0.3920	0.3751	0.3597	0.3456	0.3335
•	0.3212	0.2387	0,1724	0,1624	0.1413	0,1254	0.1131	0.1032	0.0950
k= 8									
	1	2	3	4	5	6	7	8	9
.00	0.4980	0.4960	0.4900	0.4351	0.4830	0,4803	0.4779	0.4742	0.4697
.0	0.4692	0.4414	0.4129	0.3910	0,3705	0.3530	0.3363	0.3214	0.3072
•	0.2951	0.2138	0.1706	0,1432	0,1242	0,1102	0.0992	0.0905	0.0832

Table 5.6 (continued)

k= 9	•								
	1	2	3	4	5	6	7	8	9
.00	0.4978	0.4901	0.4861	0,4831	0.4805	0.4763	0,4728	0.4634	0.4646
.0	0.4581	0.4258	0.3948	0.3728	0.3501	0.3302	0.3134	0.2979	0.2842
•	0.2719	0.1929	0.1523	0.1277	0,1107	0.0781	0.0383	0.0805	0.0740
k=10)								
	1	2	3	4	5	6	7	8	9
.00	0.4975	0.4901	0.4861	0.4806	0.4764	0.4713	0.5336	0.4622	0.4581
.0	0.4535	0.4145	0.3805	0.3541	0.3306	0.3094	0.2921	0.2765	0.2629
•	0.2509	0.1755	0.1382	0.1154	0.0998	0.0333	0.0795	0.0725	0.0666
k=11									
	1	2	3	4	5	6	7	8	9
.00	0,4973	0.4901	0.4331	0,4762	0.4711	0.4655	0.4610	0.4547	0.4507
•0	0.4494	0.4019	0.3651	0.3368	0.3118	0.2913	0.2729	0.2575	0.2443
•	0.2322	0.1605	0.1260	0.1051	0,0708	0.0303	0.0723	0.0659	0.0406
k=12									
	1	2	3	4	5	6	7	8	9
.00	0.4970	0.4859	0,4804	0.4724	0.4666	0.4606	0.4546	0,4486	0.4434
• 0	0.4344	0.3891	0.3497	0.3193	0.2938	0.2733	0.2560	0.2404	0.2276
•	0.2160	0.1478	0.1157	0.0964	0.0833	0,0737	0.0633	0.0604	0.0556
k=13									
	1	2	3	4	5	6	7	8	9
.00	0.4892	0.4823	0.4781	0,4690	0.4626	0.4550	0.4480	0.4413	0.4356
• 0	0.4286	0.3749	0.3364	0.3030	0.2785	0.2572	0.2404	0.2253	0.2126
•	0.2015	0.1370	0.1069	0.0870	0.0769	0.0680	0.0612	0.0558	0,0513
k=14									
	1	2	3	4	5	6	7	8	9
.00	0.4895	0.4827	0.4737	0.4644	0.4577	0.4491	0.4413	0.4341	0.4273
• 0	0.4233	0.3626	0.3213	0.2889	0.2633	0.2426	0.2257	0.2113	0.1992
•	0.1883	0.1273	0.0773	0.0827	0.0714	0.0632	0.0568	0.0518	0.0476
k=15									
	1	2	3	4	5	6	7	8	9
.00	0.4897	0.4797	0.4699	0.4603	0.4521	0.4427	0.4345	0.4262	0.4186
• 0	0.4128	0.3491	0.3065	0.2743	0.2493	0.2253	0.2130	0.1991	0.1871
•	0.1768	0,1191	0.0728	0.0772	0.0667	0.0570	0.0530	0.0483	0.0444

Table 5.7

Comparison of values of performance measures as predicted by the model with simulation results (D=100).

Probability of confli	ct p				
k = 3					
λ	0.1	0.3	0.5	0.7	0.9
predicted value simulation result	0.133 0.133	0.314 0.320	0.428 0.425	0.506 0.501	0.563 0.569
k = 4					
λ	0.1	0.3	0.5	0.7	0.9
predicted value Simulation result	0.165 0.162	0.356 0.356	0.453 0.459	0.533 0.527	0.584 0.586
k = 5					
λ	0.1	0.3	0.5	0.7	0.9
predicted value simulation result	0.190 0.191	0.380 0.381	0.481 0.472	0.546 0.546	0.593 0.585
k = 6					
λ	0.1	0.3	0.5	0.7	0.9
predicted value simulation result	0.210 0.213	0.395 0.397	0.490 0.486	0.551 0.551	0.597 0.588
Abort rate a					
k = 3					
λ	0.1	0.3	0.5	0.7	0.9
predicted value simulation result	0.106 0.102	0.273 0.271	0.390 0.384	0.473 0.469	0.535 0.536
k = 4					
λ	0.1	0.3	0.5	0.7	0.9
predicted value simulation result	0.142 0.130	0.332 0.328	0.444 0.446	0.520 0.515	0.574 0.575

Table 5.7 (continued)

k = 5					
λ	0.1	0.3	0.5	0.7	0.9
predicted value simulation result	0.173 0.161	0.366 0.361	0.472 0.469	0.540 0.536	0.589 0.580
k = 6					
λ	0.1	0.3	0.5	0.7	0.9
predicted value simulation result	0.197 0.186	0.387 0.388	0.485 0.483	0.5 4 9 0.5 4 6	0.595 0.597
Throughput t					
k = 3					
λ	0.1	0.3	0.5	0.7	0.9
predicted value simulation result	0.199 0.201	0.130 0.129	0.090 0.093	0.065 0.064	0.049 0.047
k = 4					
λ	0.1	0.3	0.5	0.7	0.9
predicted value simulation result	0.135 0.141	0.069 0.071	0.040 0.040	0.026 0.027	0.018 0.018
k = 5					
λ	0.1	0.3	0.5	0.7	0.9
predicted value simulation result	0.092 0.097	0.037 0.038	0.019 0.018	0.011	0.007 0.007
k = 6					
λ	0.1	0.3	0.5	0.7	0.9
predicted value simulation result	0.063 0.067	0.020 0.020	0.009 0.010	0.005 0.004	0.003 0.003

Table 5.8

Values of r and s at CC-thrashing points

<u>k</u>	<u>3</u>	44	5	6	7	8	9
CC-thrashing point	0.645	0.358	0.249	0.175	0.120	0.095	0.074
r	6.4	6.4	7.3	7.5	7.2	7.0	6.9
s	0.25	0.27	0.28	0.29	0.29	0.30	0.31
<u>k</u>	10	11	12	13	14	15	
CC-thrashing point	0.060	0.049	0.045	0.037	0.031	0.026	
r	6.9	6.6	7.8	7.4	7.0	6.1	
s	0.31	0.32	0.31	0.31	0.32	0.32	

REFERENCES

- [BG] P.A. Bernstein and N. Goodman, "Concurrency Control in Distributed Database Systems," ACM Computing Surveys, Vol. 13, No. 2, June 1981.
- [DKLPS] P.J. Denning, K.C. Kahn, J. Leroudier, D. Potier, and R. Suri, "Optimal Multiprogramming," Acta Informatica 7, 197-216 (1976).
- [G] H. Garcia-Molina, "Performance of Update Algorithms for Replicated Data in a Distributed Database," Ph.D. Thesis, Computer Science Dept., Stanford University, June 1975.
- [K] L. Kleinrock, Queueing Systems, Vol. I. (Wiley, New York, 1976).
- [L] V. Li, "Performance Models of Distributed Database Systems," Report LIDS-TH-1066, MIT Lab. for Information and Decision Systems, Cambridge, Feb. 1981.
- [LN1] W.K. Lin and J. Nolte, "Performance of Two Phase Locking," 6th Berkeley Workshop on Distributed Data Management and Computer Networks, Feb. 1982.
- [LN2] W.K. Lin and J. Nolte, "Read Only Transactions and Two Phase Locking," 2nd Symposium on Reliability in Distributed Software and Database Systems, July 1982.
- [LZ] E.D. Lazowska and J. Zahorjan, "Multiple Class Memory Constrained Queueing Networks," Proc. Performance 1983.
- [MK] R. Munz and G. Krenz, "Concurrency Control in Database Systems--A Simulation STudy," Proc. ACM SIGMOD International Conference on Management of Data, Toronto, Canada, Aug. 1977.
- [PL] D. Potier and P.L. Leblanc, "Analysis of Locking Policies in Database Management Systems," CACM 23, 10 (1980).
- [R] D. Ries, "The Effect of Concurrency Control on Database Management System Performance," Ph.D. Thesis, Computer Science Dept., University of California, Berkeley, April 1979.
- [SS] A. Shum and P. Spirakis, "Performance Analysis of Concurrency Control Methods in Database Systems," *Performance '81*, F.J. Kylstra (editor), North-Holland Publishing Company, 1981.
- [TSG] Y.C. Tay, R. Suri and N. Goodman, "Performance of Queries and Updates," manuscript in preparation.

SECTION VI

RECOVERY ALGORITHMS FOR

DATABASE SYSTEMS*

Philip A. Bernstein

Nathan Goodman

Vassos Hadzilacos

^{*}To appear in the Proceedings of IFIP '83, Paris, September 1983.

1. Introduction

A database system (DBS) processes read and write commands issued by users' transactions to access the database. If a transaction fails in midstream, or if the system fails, the database may be left in an incorrect state. For example, if a money transfer transaction fails after posting its debit but before posting its corresponding credit, then the accounts are left unbalanced. The recovery algorithm of a DBS avoids these incorrect states by ensuring that the database only includes updates that are produced by transactions that execute to completion. This paper is a survey of recovery algorithms for centralized and distributed DBS's.

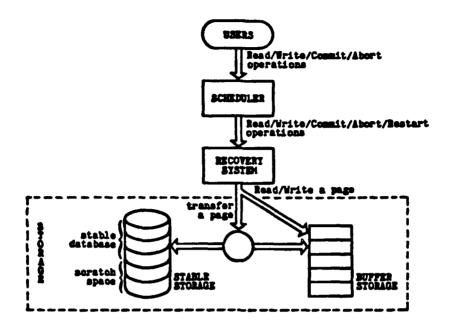
Computer systems can fail in many ways, only some of which are handled by DBS recovery algorithms. We limit our attention to clean failures in which a transaction, the system, or, in the case of a distributed DBS, one site of the system simply stops running. We do not consider traitorous failures in which components continue to run but perform incorrect actions (see [Do, PSL]). We further limit attention to soft failures in which the contents of main memory are lost, but the contents of secondary memory (disk) remain intact. We do not consider methods for recovering from disk failures, although methods similar to those in this paper apply (see [Gr, GMBLL, HR2, Li, Lo, Ve1]).

We describe a model of centralized DBS recovery in Section 2. We present four cannonical types of centralized DBS recovery algorithms in Sections 3 through 6. We describe recovery algorithms for distributed DBS's in Section 7.

2. A Model Of Centralized Database System Recovery

We model a <u>centralized database</u> system as a scheduler. a recovery

system, and storage.



Storage

The storage component consists of <u>buffer storage</u> and <u>stable</u> <u>storage</u>. Both are divided into <u>physical pages</u> of equal and fixed size. Buffer storage models main memory. Buffer storage is relatively fast, but of limited capacity, and it doesn't survive system crashes. Stable storage models disk memory and it is relatively slow, of (almost) unlimited capacity, and it does survive crashes.

The <u>latabase</u> consists of a set of <u>logical pages</u>. We assume that one physical copy (usually the most up-to-date copy) of each logical page is stored in a portion of stable storage called the <u>stable database</u>. Other portions of stable storage may be used by the recovery system as nonvolatile <u>scratch space</u> in ways that will be described later.

Transactions

A transaction is a program that can read from or write into the database. A transaction can issue four types of commands: Read, Write, Commit, and Abort. Read causes a page to be read from the database. Write causes a new copy of a logical page to be written into the database. Commit tells the system that the transaction has terminated and that all of its updated pages should be permanently reflected in the database. Abort tells the system that the transaction has terminated

abnormally and that the pages it wrote into should be returned to their previous state. (Commit and Abort may be issued by a process controlling the transaction, rather than by the transaction itself.) A transaction can have only one Commit or Abort processed.

A transaction is active if it has begun executing but has not yet had its Commit or Abort processed.

Notation: Each command is subscripted by the transaction that issued it. For example, $Read_1(P_j)$ is a Read issued by transaction T_1 on page j.

The Scheduler

The scheduler controls the order in which Reads, Writes, Commits, and Aborts are passed to the recovery system. Although the scheduler allows commands from different transactions to be interleaved, it guarantees that the resulting execution is <u>serializable</u>. An execution is serializable if the effect is exactly the same as if the transactions had been executed serially, one after the next, with no concurrency at all. Many scheduling algorithms for attaining serializability are known [BG1, 2]; versions of all of them are compatible with the recovery algorithms described in this paper.

The scheduler also guarantees that the execution is <u>recoverable</u>. An execution is recoverable if, for each transaction T_i , T_i is not committed until, for each page read by T_i , the transaction that last wrote that page is committed.

Recoverability is needed to avoid errors such as the following. Suppose T_i reads a page P_k last written by T_j (which is still active), T_i writes another page P_l , and commits. Now, suppose T_j fails and is aborted. Aborting T_j causes its write on P_k to be undone, thereby rendering T_i 's input invalid. But, since T_i cannot be aborted after having been committed, T_i 's updates to P_l must remain in the database even though its input P_k is invalid.

For definitions, we assume that the scheduler uses <u>page-level two-phase locking (2PL)</u> [EGLT]. Before outputting $Read_1(P_j)$ (resp. Write₁(P_j)), the scheduler sets a read lock (resp. write lock) on page P_j for transaction T₁. Two transactions cannot concurrently own <u>con-</u>

flicting locks on the same page, where read locks conflict with write locks and write locks conflict with read and write locks. If the scheduler receives an operation for which it can't set the corresponding lock, it delays the operation until the lock can be set.

When the scheduler receives a $Commit_1$ or an $Abort_1$, it forwards the operation directly to the recovery system. When the recovery system acknowledges that the operation has been processed, the scheduler then releases all the locks held by T_4 .

Two-phase locking ensures serializability (see [BG2, EGLT] for proofs). The version of 2PL presented above also ensures recoverability by requiring that a transaction hold its write locks until its Commit or Abort is processed.

The Recovery System

The recovery system processes the Read, Write, Commit, and Abort commands it receives from the scheduler. It also handles system failures.

A system failure can interrupt the DBS at any moment. It causes all processing to stop and the contents of buffer storage to be lost. After the system recovers, transactions that were active at the time of the failure cannot continue executing because the contents of main memory are now useless. Thus, after the failure and before processing any other commands, the recovery system processes the <u>restart</u> command, whose effect is to abort all active transactions.

To handle failures properly, it is essential that the Commit command be implemented in a single instruction, normally a page write. If it were to require more than one instruction, a system failure could interrupt a partially completed Commit, making it ambiguous whether the transaction should be aborted during restart. Said differently, each transaction must always be in one of three states: active, committed, or aborted, and each state change must be implemented by an atomic instruction execution.

Structuring Scratch Space

There are several types of information that a recovery algorithm stores in stable scratch space. It may store the identifiers of

transactions that have committed, called the <u>commit list</u>. In this case, the single instruction that implements Commit_i is usually a write that adds T_i to the commit list. The recovery algorithm may also store a list of identifiers of transactions that are active, called the <u>active</u> <u>list</u>, and those that have aborted, called the <u>abort list</u>.

Recovery algorithms often store copies of pages that were recently written on an audit trail (sometimes called a journal or log). For each write processed by the recovery algorithm, the audit trail may contain the identifier of the transaction that performed the write, a copy of the newly written page (called an after-image), and a copy of the physical page in the stable database that was overwritten by the write (called a before-image). Different algorithms vary considerably in the information they keep on the audit trail and in how they structure that information.

Undo and Redo

Recovery algorithms also differ in the time at which they write pages into the stable database. They may perform such writes before, concurrently with, or after the atomic instruction that commits the transaction that last wrote those pages.

Suppose that a page statten by an active transaction is written into the stable database before the transaction commits. If the transaction aborts due to a system of transaction failure, the recovery algorithm must undo the write by restoring the previous copy (before-image) of the page

Suppose that a page written by an active transaction is <u>not</u> written into the stable database before the transaction commits. If a system failure occurs after the transaction commits but before the page is written into the stable database, the recovery algorithm must <u>redo</u> the write by moving the page to the stable database.

In every recovery algorithm, the after-images produced by a transaction must be written to stable storage (the database or scratch space) before the transaction commits. This is called the commit rule. If it is violated, a system failure shortly after a transaction T_1 commits could leave the recovery algorithm with no stable copy of T_1 's after-images, making it impossible to redo T_1 .

Every recovery algorithm must also obey the <u>log ahead rule</u>: if an after-image is written to the stable database before the transaction that wrote it commits, then the before-image of that page must first be written to the audit trail. Otherwise, a system failure could occur after the after-image is in the stable database but before the before-image is in the audit trail, in which case the write could not be undone.

Categorization of Recovery Algorithms

Recovery algorithms can be categorized based on the timing of updates to the stable database. There are four types of recovery algorithms: ones that require undo but not redo, redo but not undo, both undo and redo, and neither undo nor redo. These types of algorithms are described in Sections 3-6.

3. Algorithms That Undo But Don't Redo

For each type of recovery algorithm, we present a generic algorithm based on our database system model and then we list example implementations. We describe this generic version by explaining how each command is processed. In all of the algorithms, the first command processed for T_1 should add T_1 to the active like.

For each operation, we mark by "{Ack}" the point at which the recovery system can acknowledge to the scheduler that the operation has been completed. Sometimes the operation has additional work to do after the acknowledgement is sent.

 $Read_1(P_1)$. Copy P_1 from the stable database into a buffer. {Ack}

Write₁(P_j). Copy the before-image of P_j (from the stable database) to the audit trail. {Ack} Then* (after the disk acknowledges the write in the audit trail), write the new copy of P_j into the stable database.

In every algorithm, we use "then" to mean "wait for the previous ste; to complete before projecting to the next step".

DISTRIBUTED DATABASE CONTROL AND ALLOCATION VOLUME 1 FRAMENORKS FOR UNDER. (U) COMPUTER CORP OF AMERICA CAMBRIDGE MA N K LIN ET AL OCT 83 RADC-TR-83-226-VOL-1 F30602-81-C-0028 F/G 9/2 AD-A138 891 3/4 UNCLASSIFIED NL



MICROCOPY RESOLUTION TEST CHART NATIONAL BIJREAU OF STANDARDS-1963-A

 $Commit_1$. Make sure all pages written by T_1 are in the stable database. Then write T_1 into the commit list. {Ack} Then delete it from the active list.

Abort₁. Write T_1 into the abort list. Then undo all of T_1 's writes by reading their before-images from the audit trail and writing them back into the stable database. {Ack} Then, delete T_1 from the active list.

Restart. Process Abort; for each Ti on the active list. {Ack}

In this algorithm, all pages written by a transaction are written into the *table database before the transaction commits. Thus, redo is never needed, but an abort may require undo.

It is actually not necessary to write an after-image into the stable database <u>immediately</u> after the before-image is written into the audit trail. The after-image could be left in buffer storage for awhile, provided it is written to the audit trail before the transaction commits as required by the commit rule.

This algorithm obeys the log ahead rule in processing $Write_{i}(P_{j})$; the before-image of P_{j} is written to the audit trail before the afterimage is written to the stable database.

The order in which writes are applied to stable storage is quite sensitive in this (and most other) recovery algorithms. In this algorithm, for example, in processing commit₁ it is incorrect to delete T_1 from the active list before writing it into the commit list.

Remember that a system failure can occur during the processing of a Restart. So Restart must also take care to write pages to stable storage in order that it will be resilient to an system failure (followed by another Restart).

After $Commit_1$ or $Abort_1$ has been processed, the audit trail copies of pages written by T_1 are no longer needed and can be returned to free space. The algorithm for garbage collecting these audit trail pages depends principally on the audit trail's data structure. We will not discuss garbage collection issues for any of the recovery methods described in this paper.

The Prime Algorithm

This type of recovery algorithm is used in a database system product offered by Prime Computers [Du], and the Adaplex database system being developed at CCA [CFLNR].

In Prime's algorithm, each page in the stable database has a pointer to its before-image in the audit trail. Each before-image in the audit trail points, in turn, to the next older before-image of the same page. Also, each physical page carries the transaction identifier of the transaction that wrote that particular copy. And, for each active transaction there is a convenient way to obtain a list of all pages it has written.

The page pointers are used for two purposes. First, to process an Abort, the pointer in each stable database page makes it easy to undo the aborted transaction's writes. Second, they help avoid concurrency control conflicts between queries and updates, as follows.

A <u>overy</u> is a read-only transaction. Reads issued by queries are not locked in the scheduler but are passed directly to the recovery system (without being delayed). When the recovery algorithm receives the <u>first</u> read issued by a query T₁, say Read₁(P_j), it reads the commit list and then selects the newest copy on the chained list of P_j copies whose transaction identifier is on the commit list. Subsequent reads by T₁ are processed in the same way, using the copy of the commit list that was read when the first Read₁ was processed. By reading in this way, queries see a consistent copy of the database, yet, they do not set read locks that might delay update transactions.

Another undo/no-redo algorithm is described in [Ra].

4. Algorithms That Redo But Don't Undo

In the generic algorithm, each command is processed as follows.

 $Read_1(P_j)$. If T_1 previously wrote P_j , then copy the after-image of P_1 into a buffer. Otherwise, copy P_1 from the stable database into a

buffer. {Ack}

Write₁(P_j). Write the new value of P_j into the audit trail. {Ack}

 $Commit_1$. Write T_1 into the commit list: Then for each page written by T_1 , copy the after-image from the audit trail into the stable database. {Ack} Then delete T_1 from the active list.

Abort₁. Write T_1 into the abort list. {Ack} Then delete it from the active list.

Restart. For each T_1 that is on the active list but not on the commit list, process Abort₁. {Ack} For each T_j on the active list and the commit list, process Commit₁.

In this algorithm, pages written by a transaction are not written into the stable database until after the transaction commits. Thus, undo is never needed, but a Restart may require redo.

This algorithm obeys the commit rule, because the after-image of pages written by T_1 are stored on the audit trail before T_1 commits. It also obeys the log shead rule, since no after-image of a transaction is written into the stable database before it commits.

Implementations of this algorithm are described in [LS, ML]. This type of recovery algorithm is used in the Ingres Database System [St] and in SDD-1 [BSR].

5. Algorithms That Redo And Undo

In this algorithm, commands are processed as follows.

Read₁(P_j). If T₁ previously wrote P_j, then copy the after-image of P_j into a buffer. Otherwise, copy P_j from the stable database into a buffer. {Ack}

Write₁(P_j). Copy the before-image and the after-image of P_j into the audit trail. {Ack} Then, sometime later, write the after-image into the stable database.

 $Commit_1$. Write T_1 into the commit list. Then, for each page written by T_1 , write the after-image into the stable database (if it hasn't already been done). {Ack} Then, delete T_1 from the active list.

Abort₁. Write T_1 into the abort list. Then, for each page written by T_1 , if its after-image has already been written into the stable database, write its before-image into the stable database. {Ack} Then delete T_1 from the active list.

Restart. For each T_1 on the active list and the commit list, process $Commit_1$. For each T_1 on the active list but not on the commit list, process $Abort_1$. $\{Ack\}$

Note that Abort may require undo and Restart may require redo.

This algorithm obeys the commit rule, since the after-image of each page written by T_1 is written into the audit trail before T_1 commits. It also obeys the log ahead rule, since the before-image of each page written by T_1 is written into the stable database.

One can improve the performance of this algorithm by using a variation proposed by Gray [Gr]. Gray's algorithm processes commands as follows.

 $Read_1(P_j)$. If T_1 previously wrote P_j , check to see if the afterimage is in buffer storage. If not, copy P_j from the stable database to a buffer. {Ack}

Write₁(P_j). Copy the before-image of P_j into buffer storage unless it is already there. Write the after-image of P_j into buffer storage; this step must not overwrite the before-image. {Ack} Sometime later, write the before-image into the audit trail, leaving a copy of the after-image in buffer storage. The after-image may be written into the stable database any time after the before-image is written into the audit trail. Once the after-image is written both to the audit trail and the stable database, it may be removed from buffer storage.

 $Commit_1$. After all the after-images of pages written by T_1 have been written into the audit trail, write T_1 into the commit list. {Ack}

Abort; and Restart are the same as the generic algorithm.

This algorithm obeys the log shead rule because the before-image of each page is written in the audit trail before the after-image is written in the stable database. The commit rule is also satisfied since T_1 's after-images are written into the audit trail before T_1 commits.

When all after-images written by T_i have been written into the stable database, T_i can be deleted from the active list. This tells Restart that T_i does not need to be redone.

The main benefit of this algorithm is that the decision to write pages into stable storage is usually left to the database system's buffer management algorithm. The recovery algorithm writes into stable storage only when the commit or log ahead rule requires it.

A detailed implementation of this algorithm which incorporates checkpoints and in which transactions write records instead of entire pages appears in [Li].

6. Algorithms That Don't Undo Or Redo

In the generic algorithm, each command is processed as follows.

 $Read_1(P_j)$. If T_1 previously wrote P_j , then copy the after-image of P_1 into a buffer. Otherwise, copy P_j from the stable database into a

buffer. {Ack}

Write₁(P_j). Write the after-image of P_j into the audit trail. {Ack}

Commit_i. In a single instruction, write the after-images of all pages written by T_i into the stable database and delete T_i from the active list. {Ack}

Abort_i. Write T_i into the abort list. {Ack} Then delete it from the active list.

Restart. For each Ti on the active list, process Aborti. {Ack}

Unfortunately, this description isn't very informative because it relies on a magical instruction that implements commit without even using a commit list. Notice that if the magical instruction is available, then undo isn't needed because a transaction's after-images are not written into the stable database before it commits, and redo isn't needed because a transaction's after-images are written into the stable database in the instruction that commits the transaction.

We will describe an implementation of the Commit instruction similar to one presented in [Lo].

Lorie's Shadow Page Algorithm

Assume that the stable database is partitioned into files $\{F_1,\ldots,F_2\}$, each of which is a sequence of logical pages. Each file, F_j , has a page table, PT_j , whose entries point to the pages of F_j . That is, $PT_j[k]$ contains the address of the k-th page of F_j ; this page is denoted P_{jk} . Assume that each page table fits on one page in the stable database. The stable database also contains in a fixed address a master record, H_j , that points to the n page tables; H[j] contains the address of PT_j .

Abort and Restart are processed as in the generic algorithm. Read, Write, and Commit are processed as follows.

For each file, F_j , the first Read or Write that T_i issues on a page of F_j causes the recovery algorithm to make a copy of PT_j in buffer storage, denoted PT_{ji} . For each page P_{jk} that T_i writes, $PT_{ji}[k]$ will

point to the after-image of that page in the audit trail. (The other entries in $PT_{\underline{1}\underline{1}}$ are irrelevant.)

 $Read_1(P_{jk})$. If T_1 previously wrote P_j , then copy the after-image of P_j from address $PT_{jk}[k]$ into a buffer. Otherwise, use M to find PT_j and copy P_{jk} from address $PT_j[k]$ in the stable database into a buffer. {Ack}

 $Write_i(P_j)$. Write the new copy of P_{jk} into the audit trail. Then, assign $PT_{ij}[k]$ the address of that audit trail page. {Ack}

Commiti. Copy M into buffer storage. For each file Fj that Ti wrote into, use (the buffer copy of) M to find PTj and copy it into an empty page of buffer storage. (There are now two page tables for Pj connected to Ti: the buffer copy of PTj that was just read and PTji.) For each page Pjk that was written by Ti, assign to the buffer copy of PTj[k] the contents of PTji[k]. Then, write PTj into a new location in scratch space; denote this new copy of PTj by PTj. Then, for each Fj that Ti wrote into, assign to (the buffer copy of) M[j] the address of PTj. Then write M back to its fixed address in stable storage. {Ack}

The commit algorithm prepares a scratch copy of the page table (PT_j) . This is accomplished by assigning to M[j] the address of PT_j for each file F_j that T_i wrote. By writing M back to the stable database, the old copies of the page table (PT_j) are replaced by the new ones (PT_j) .

The instruction that commits T_1 is the one that writes the updated M back into the stable database. Before this write, any read will use the old copy of M to read the before-image of any page written by T_1 . After this write, it will read the after-image of any such page.

The recovery algorithm can only commit one transaction at a time. That is, Commit is a critical section. If two transactions were (incorrectly) to commit concurrently, each transaction might read a copy of PT_j into buffer storage, change the pointers to pages it wrote, and write that copy of PT_j to the audit trail. Thus, two copies of PT_j would exist. Whichever transaction updated M first would lose its updates to PT_j, since they would be overwritten by the second transaction when it installed <u>its</u> copy of PT_j by updating M.

A version of Lories's algorithm is implemented in System R's, recovery manager [GMBLL].

7. Recovery In A Distributed Database System

A distributed database system (DDBS) consists of a set of sites connected by a network. Each transaction can read or write data stored at any of the sites.

We model a DDBS by a set of processes called <u>data modules</u> (DMs) and <u>transaction modules</u> (TMs). A DM is a centralized database system as defined in Section 2. It processes Reads and Writes on pages stored at that DM. It also processes Commits and Aborts, which permanently install or undo the writes of a transaction at that DM.

A TM interfaces transactions and DMs. Each transaction, T_1 , submits commands to one TM, say TM_2 . To process $Read_1$ or $Write_1$, TM_2 simply sends the command to the DM that stores the data being read or $Write_1$ ten. Let $Active_1$ be the DMs at which T_1 was active. To process $Abort_1$, TM_2 must ensure that every DM in $Active_1$ processes $Abort_1$. To process $Commit_1$, TM_2 should try to ensure that every DM in $Active_1$ processes $Commit_1$.

Unfortunately, TMs and DMs may fail at unpredictable times. TM_a must process commands so that such failures never cause it to produce incorrect results.

We assume that process (i.e., TM and DM) failures are "clean". If a process does not produce an expected response to a message within a timeout period, then the process has <u>really</u> failed. If one process believes another process is down, then <u>all</u> processes believe that the process is down. And, when a process recovers, it recognizes that it has just recovered from a failure and runs a special "reintegration protocol". Hechanisms that support these assumptions are beyond the scope of this paper. (See [ABG, HS, FR, Wa].)

Each TM keeps an active list, commit list, and abort list in stable storage. And, for each T_1 on the active list, it maintains Active in stable storage. When it receives a Read or Write from T_1 , it sends the command to the appropriate DM and adds that DM to Active. For the first such Read or Write, the TM also adds T_1 to its active list. It processes Abort and Commit as follows.

Abort_i. Add T_i to the abort list. Then, send Abort_i to each DM in Active_i. Wait for every DM to acknowledge that it processed Abort_i. {Ack} Delete T_i from the active list.

Commit₁. Add T₁ to the commit list. Then, send Commit₁ to each DM in Active₁. Wait for every DM to acknowledge that it processed Commit₁. {Ack} Delete T₁ from the active list.

If a TM fails and later restarts, then it processes a Restart in the usual way: For each T_1 on both the commit list and the active list, process Commit₁. For all other T_1 on the active list, process Abort₁.

If a TM, say TM_a, discovers that a DM, say DM_b, has failed, then it normally processes Abort₁ for each T₁ that has DM_b in Active₁. But what if DM_b is in Active₁ and TM_a has already sent Commit₁ to other DMs in Active₁? In this case it can't abort T₁, because T₁ may already be committed at some DMs. Instead, it must wait for DM_b to recover. When it does, TM_a sends Commit₁ to DM_b too.

Two-Phase Commit

Each TM must obey the commit rule. That is, it must not send Commit_i to any DM until every DM in Active_i has T_i's after-images on stable storage. Otherwise, a DM in Active_i may:

- 1. Fail before receiving Commiti.
- 2. Upon recovering, discover from TM_a that T_i has committed.
- 3. But be unable to process $Commit_1$ because it lost some of T_1 's after-images due to the failure.

To obey the commit rule and thereby prevent (3), TM_a can use the <u>two-phase commit</u> protocol for processing Commit commands [LS]. Phase one begins when TM_a receives Commit₁. It then sends a command called

End₁ to each DM in Active₁. A DM processes End₁ by first ensuring that T_1 's after-images at that DM are on stable storage and then sending an acknowledgement to TM_a . When TM_a has received the acknowledgement from every DM in Active₁, phase one is done. {Ack} In phase two, TM_a sends Commit₁ to each DM.

Since TM_a does not send $Commit_1$ to any DM until every DM has acknowledged End_1 , no DM in Active will process $Commit_1$ until every DM has T_1 's after-image on stable storage.

If a DM, say DM_b, fails before acknowledging End_1 , then TM_a won't leave phase one. Since TM_a cannot be sure that DM_b will be able to process Commit_1 when it recovers, TM_a must either wait for DM_b to recover or abort T_1 by sending Abort₁ to every DM in Active₁. In practice, TM_a simply waits a prespecified timeout period after distributing the End_1 's; if it hasn't received an acknowledgement of some End_1 by this time, it assumes the DM has failed and aborts T_1 .

Until a DM processes End_i , it may unilaterally decide to abort T_i by sending an Abort_i command to TM_a . Once a DM acknowledges End_i , it loses its right to unilaterally abort T_i , and may only abort T_i if directed to do so by TM_a .

Three-Phase Commit

CARL CARLES OF THE STATE OF THE

The TM algorithm presented above has a serious disadvantage. Suppose TM_a sends End_i to DM_b, DM_b acknowledges End_i, and then TM_a fails. Since DM_b doesn't know whether T₁ will commit or abort, it has to wait for TM_a to recover. In particular, it must hold T₁'s locks until TM_a recovers. If TM_a is supervising many active transactions, large portions of the database may be locked and unavailable until TM_a recovers.

We can avoid this problem by providing each TM with one or more backup TMs. If a TM fails, the backups can take over its functions.

One such algorithm is <u>three-phase commit</u> [Sk1, 2, 3, SS]. Each backup for TM_R maintains a commit list, CL_R . To process Commit TM_R behaves as follows.

- 1. The sends End; to each DM in Active;. Then, it waits for all DMs to acknowledge their End; 's.
- 2. TM_a sends a command called Precommit₁ to each backup TM. A TM processes Precommit₁ by adding T₁ to its copy of CL_a, and then sending an acknowledgement to TM_a. TM_a waits for all backups to acknowledge Precommit₁.
- 3. TMa sends Commiti to each DM in Active:

Essentially, this is the two-phase commit protocol w a new phase added (step (2)).

If a backup TM fails, TM_a can ignore the failure if the number of backups is still acceptably large; otherwise, it should acquire another backup TM to replace the failed one.

Suppose TM_a fails. When the backups discover the failure, they elect one of their member TMs, say TM_b , to replace TM_a . After TM_b is elected, every other backup TM sends its copy of CL_a to TM_b . TM_b takes the union of those copies and distributes the result to other backups. This becomes everyone's copy of CL_a . When this process is complete, TM_b tells all DMs that it has taken over TM_a 's functions.

If a DM wants to know what happened to a particular transaction, T_1 , that was supervised by TM_a , it asks TM_b . If T_1 is in TM_b 's CL_a , then TM_b tells the DM to commit T_1 ; otherwise, it tells the DM to abort T_1 . Thus, a transaction that was supervised by TM_a is committed if and only if it reached the second phase of three-phase commit and at least one of its precommits reached a backup TM (that didn't fail).

The algorithm for electing a backup TM to replace TM_a is easy, as long as none of the backups fail or recover from failure during the election. Assume each TM has a unique identifier. To elect a replacement for TM_a , each backup exchanges its identifier with every other backup. The TM with the largest identifier wins the election and takes over.

If backup TMs fail or recover from failure during the election, the above algorithm can misbehave. Each of two TMs can conclude that it won the election. Algorithms to prevent this behavior are discussed in [Ga, SS].

It is possible that TM_a and all of its backups fail during a short time period — too short for replacement backups to be acquired. This is called a <u>total failure</u> of TM_a ; no TM can ever take over its function. DMs must wait until TM_a and enough of its backups have recovered that the corrects status of TM_a 's transactions can be determined. Algorithms for recovering after total failure are discussed in [Sk1].

Many variations on three-phase commit protocols have been proposed and analyzed. See [ABDG, AD, Co, Ea, HS, La, MPM, TGGL].

Replicated Data

If a DM fails, transactions that need that DM's data must wait for the DM to recover. To avoid this delay, the DBS can replicate data; that is, it can store parts of the database at more than one DM. If one copy is unavailable due to a DM failure other copies can be used instead.

Many concurrency control algorithms are known for keeping multiple copies of each page mutually consistent. However, even if concurrency control is performed correctly, failures can cause transactions to malfunction.

For example, suppose P_1 has copies P_{1a} and P_{1b} at IM_a and IM_b (resp.), and P_2 has copies P_{2c} and P_{2d} at IM_c and IM_d . T_1 reads P_1 and writes P_2 ; T_2 reads P_2 and writes P_1 . Asplicated data is handled by the "intuitive" algorithm: to read data, read any copy; to write data, write all available copies. The following execution obeys these rules, yet it is incorrect.

Read₁(P_{1a})
$$\longrightarrow$$
 DM_d-fails \longrightarrow Write₁(P_{2c})
Read₂(P_{2d}) \longrightarrow DM_a-fails \longrightarrow Write₂(P_{1b})

This execution is incorrect because T_1 reads (a copy of) P_1 before T_2 writes P_1 , while T_2 reads (a copy of) P_2 before T_1 writes P_2 . The first condition means that T_1 appears to precede T_2 , while the second condition means that T_2 appears to precede T_1 . These conditions cannot both hold in a serial execution, and so the given execution is incorrect.

Algorithms for correctly processing commands on replicated data in the presence of DM failures appear in [ABDG, AD, Ea, Gi, HS, MPM, Th].

No consensus on the best approach to this problem has yet emerged.

8. Network Partitions

A partition is a communication failure that splits the network of sites into two or more subnetworks such that each site in one subnetwork is unable to communicate with any site in any other subnetwork (besides its own). After the partition, the DDBS must decide how to continue with transactions that were active at the time of the partition and those that begin executing after the partition.

The latter transactions are easy to handle. A transaction can be processed normally provided that it reads and writes pages that reside in one communicating subnetwork and that none of the pages it writes have replicated copies of the subnetwork. Otherwise, the transaction cannot be processed at all.

difficult to handle. If a transaction was (and needs to be) active only at sites in one communicating subnetwork, then it can continue being processed normally. If a transaction had not yet processed its End; at some site in a subnetwork, then the transaction can be aborted in that subnetwork. If all sites in a communicating subnetwork have processed End; and one or more have also processed Commit; (resp. Abort;), then the transaction can be committed (resp. aborted) at all sites in that subnetwork. If all sites in a subnetwork have processed End; but none have processed Commit; or Abort; then the transaction is stuck. This subnetwork cannot

determine whether the transaction was committed or aborted in some other subnetwork with which it cannot communicate. Thus, it must leave the transaction in an active but blocked state, until the partition is repaired.

Avoiding this blocking situation has been the subject of much discussion and research. The probability of this event can possibly be decreased by careful database design and careful selection of backup TM's for three-phase commit. However, the situation apparently cannot be eliminated entirely [Skl].

9. References

- [ABDG] Alsberg P.A., G.G. Belford, J.D. Day, and E. Grapa. "Multi-copy Resiliency Techniques," <u>Distributed Data Management</u> (J.B. Rothnie, P.A. Bernstein, D.W. Shipman, eds.), IEEE, 1978, pp. 128-176.
- [ABG] Attar R., P.A. Bernstein, and N. Goodman. "Site Initialization, Recovery. and Back-up in a Distributed Database System," <u>Proc. 6th Berkeley Workshop</u>, Feb. 1982, pp. 185-202.
- [AD] Alsberg, P.A., and J.D. Day. "A Principle for Resilient Sharing of Distributed Resources," <u>Prog. 2nd Intl. Conf. Software Eng.</u>, Oct. 1976.
- [ADEH] Andler, S., I. Ding, K. Eswaran, C. Hauser, W. Kim, J. Mehl, R. Williams. "System D: A Distributed System for Availability,"

 Proc. 8th VLDB, Sept. 1982, pp. 33-44.
- [Ba] Bartlett, J.F. "A 'NonStop' Operating System," in <u>The Theory and Practice of Reliable System Design</u>, (Siewiarek and Swarz, eds.), Digital Press, 1982, pp. 453-460.
- [Bj] Bjork, L.A. "Recovery Scenario for a DB/DC System," Proc. ACM Nat'l Conf., 1973, pp. 142-146.
- [BD] Bjork L.A., and C.T. Davies. "The semantics of the preservation and recovery of integrity in a data system," IBM TR-02.540, Dec. 22, 1972.
- [BG1] Bernstein, P.A., and N. Goodman. "Concurrency Control in Distributed Database Systems," <u>ACM Computing Surveys</u> 13, 2 (June 1981), pp. 185-221.
- [BG Bernstein, P.A., and N. Goodman. "A Sophisticate's Introduction to Distributed Database Concurrency Control," <u>Proc. 8th VLDB</u>, Sept. 1982, pp. 62-76.

- [BHR] Bayer, R., H. Heller, and A. Reiser. "Parallelism and Recovery in Database Systems," <u>ACM Trans. on Database Systems</u>, Vol. 5, No. 2 (June 1980), pp. 139-156.
- [BSR] Bernstein, P.A., D.W. Shipman and J.B. Rothnie, "Concurrency Control in a System for Distributed Databases (SDD-1)," <u>ACM Trans on Database Sys.</u> 5, 1 (March 1980), pp. 18-51.
- [Co] Cooper, E.C. "Analysis of Distributed Commit Protocols," <u>Proc.</u>

 <u>ACM SIGMOD Conf. on Management of Data</u>, ACM, June 1982, pp. 175183.

- [CB] Cheng, W.K. and G.G. Belford. "The Resiliency of Fully Replicated Distributed Databases," <u>Proc. 6th Berkeley Workshop</u>, Feb. 1982, pp. 23-44.
- [CFLNR] Chan, A., S. Fox, T.A. Landers, A. Nori, and D. Ries. "The Implementation of an Integrated Concurrency Control and Recovery Scheme." Proc. ACM SIGMOD Conf. on Management of Data, June 1982, pp. 184-191.
- [Da] Davies, C.T. "Recovery Semantics for a DB/DC system," Proc. ACM Nat'l Conf., 1973, pp. 136-141.
- [Do] Dolev, D. "The Byzantine Generals Strike Again," J. of Algorithms, 3, 1 (1982).
- [Du] Dubourdieu, D.J., "Implementation of Distributed Transactions,"

 Proc. Sixth Berkeley Workshop on Distributed Data Management and

 Computer Networks, 1982, pp. 81-94.
- [Ea] Eager, D.L. "Robust Concurrency Control in a Distributed Database." Univ. of Toronto TR CSRG #135, Oct. 1981.
- [EGLT] Eswaran, K.P., J.N. Gray, R.A. Lorie, and I.L. Traiger. "The Notions of Consistency and Predicate Locks in a Database System,"

 <u>Commun. ACM</u>, Vol. 19, No. 11, Nov. 1976, pp. 624-633.
- [FM] Fischer, M.J. and A. Michael. "Sacrificing Serializability to Attain High Availability of Data in an Unreliable Network," <u>Proc. 1st ACM SIGACT-SIGMOD Symp. on Principles of Database Systems</u>, ACM, Mar. 1982, pp. 70-75.
- [Ga] Garcia-Molina, H. "Elections in a Distributed Computing System,"

 IEEE Trans on Computers C-31, 1(Jan. 1982), pp. 48-59.
- [Gi] Gifford, D.K. "Weighted Voting for Replicated Data," <u>Pron. 7th</u>

 <u>Symp. on Operating Systems Principles</u>, ACM, Dec. 1979, pp. 150
 159.

- [Gr] Gray, J.N. "Notes on Database Operating Systems," in <u>Operating</u>

 <u>Systems</u>: an <u>Advanced Course</u>, Springer-Verlag, 1979.
- [GMBLL] Gray, J.W., P. MoJones, M. Blasgen, B. Lindsay, R. Lorie, T. Price, F. Putzulo, and I. Traiger. The Recovery Manager of the System R Database Manager, ACM Computing Surveys, 13, 2 (June 1981), pp. 223-242.
- [HR1] Harder, T., and A. Reuter. "Optimization of logging and recovery in a database system," in <u>Database Architecture</u>, Bracchi and Nijssen eds., North-Holland, 1979, pp. 151-168.
- [HR2] Harder, T., and A. Reuter. "Principles of Transaction Oriented Database Recovery -- A Taxonomy," Univ. Kaiserslautern TR 50/82.
- [HS] Hammer, M.M., and D.W. Shipman. "Reliability Mechanisms for SDD-1: A System for Distributed Databases," <u>ACM Trans. on Database Syst.</u>, Vol. 5, No. 5 (Dec. 1980), pp. 431-466.
- [Kim] Kim, K.H. "Error Detection, Reconfiguration and Recovery in Distributed Processing Systems," <u>Conf. on Distributing</u>, IRRE, 1979, pp. 284-294.
- [La] Lamport, L. "The Implementation of Reliable Distributed Multiprocess Systems," Computer Networks, I 2 (1978), pp. 95-114.
- [Li] Lindsay, B.G. at al. "Notes on Distributed Databases," IBM Research Report, No. RJ2571, July 1979.
- [Lo] Lorie, R.A. "Physical Integrity in a Large Segmented Database,"

 ACM Trans. on Database Syst., Vol. 2, No. 1 (Mar. 1977), pp. 91104.
- [LS] Lampson, B.W. and H. Sturgis, "Crash Recovery in a Distributed Storage System," Technical Report, Xerox PARC, 1976
- [LSP] Lamport, L., R. Shostak, and M. Pease. "The Byzantine Generals Problem." ACM Trans. on Programming Languages and Systems, Vol. 4, No. 3 (July 1982), pp. 382-401.
- [ML] Menasce, D.A., and O.E. Landes. "On the Design of a Reliable Storage Component for Distributed Database Management Systems,"

 <u>Proc. 6th VLDB</u>, Oct. 1980, pp. 365-375.
- [MPM] Menasce. D.A., G.J. Popek, and R.R. Muntz. "A Locking Protocol for Resource Coordination in Distributed Databases," <u>ACM Trans.</u> on <u>Database Syst.</u>, Vol. 5, No. 2, (June 1980), pp. 103-138.
- [PR] Parker, D.S., and R.A. Ramas. "A Distributed File System Architecture Supporting High Availability," Proc. 8th YLDB, Sept. 1982, pp. 161-184.

- [PSL] Pease, M., R. Shostak, and L. Lamport. "Reaching Agreement in the Presence of Faults," JACM, 27, 2 (1980), pp. 228-234.
- [Ra] Rappaport, R.L. "File Structure Design to Facilitate On-Line Instantaneous Updating," Proc. of the 1975 SIGMOD Conf., pp. 1-14.
- [Ree] Reed, D.P. "Implementing Atomic Actions," <u>Proc. 7th ACM Symp. on Operating Systems Principles</u>, ACM, Dec. 1979.
- [Reu] Reuter, A. *A Fast Transaction-Oriented Logging Scheme for Undo Recovery.* IEEE Trans on Soft. Eng., SE-6 (July 1980), pp. 348-356.
- [Sc] Schlageter, G. "Enhancement of Concurrency in DBS by the Use of Special Rollback Methods," DB Architecture, Bracchi and Nijssen eds., North-Holland, 1979, pp. 141-149.
- [Sk1] Skeen, D., "Crash Recovery in a Distributed Database System," Ph.D. Thesis, Dept. of Elec. Eng. and Comp. Sci., Univ. of California, Berkeley, 1981.
- [Sk2] Skeen, D. "Nonblocking Commit Protocols," <u>Proc. 1982 ACM-SIGMOD</u>

 <u>Conf. on Management of Data, ACM, pp. 133-147.</u>
- [Sk3] Skeen, D. "A Quorum Based Commit Protocol," <u>Proc. 6th Berkeley</u>
 Workshop, Feb. 1982, pp. 69-80.
- [St] Strom. B.I. "Consistency of Redundant Databases in a Weakly Coupled Distributed Computer Conferencing System," Proc. 5th Berkeley Workshop, 1981, pp. 143-153.
- [SS] Skeen, D., and M. Stonebraker. "A Formal Model of Crash Recovery in a Distributed System," Proc. 5th Berkeley Workshop, 1981, pp. 129-142.
- [St] Stonebraker, M. "Concurrency Control and Consistency of Multiple /copies of Data in Distributed INGRES," IEEE Trans. on Soft. Eng., SE-5, 3 (May 19779), pp. 188-194.
- [TGGL] Traiger. I.L., J. Gray, C.A. Galtier, and B.G. Lindsay. "Transactions and Consistency in Distributed Database Systems," <u>ACM</u>

 <u>Trans. on Database Systems</u>, Vol. 7, No. 3, (Sept. 1982), pp. 323-342.
- [Th] Thomas, R.H. "A Majority Consensus Approach to Concurrency Control for Multiple Copy Databases," <u>ACM Trans. on Database Syst.</u>, 4, 2 (June 1979), pp. 180-209.

- [Ve1] Verhofstad, J.M.S. "Recovery Techniques for Database Systems,"

 ACM Computing Surveys, 10, 2 (1978), pp. 167-196.
- [Ve2] Verhofstad, J.M.S. "Recovery Based on Types," DB Architecture,
 Bracchi and Nijssen eds., North-Holland, 1979, pp. 125-139.
- [Wa] Walter, B. "A Robust and Efficient Protocol for Checking the Availability of Remote Sites," <u>Proc. 6th Berkeley Workshop</u>, Feb. 1982, pp. 45-68.

SECTION VII

AN OPERATIONAL MODEL FOR

DATABASE SYSTEM RELIABILITY

Vassos Hadzilacos

1. INTRODUCTION

1.1 Overview

In this paper we develop an operational (i.e. state-based) model for studying reliability of database systems (dbs). The system is described at any point in time by a "system state". Reliability-related properties of the system (e.g. "resiliency") can be expressed as predicates on the system state. Transaction processing algorithms (hereafter called simply "algorithms") can be described as state transition functions, mapping the current system state to the next. Finally, correctness and other reliability properties of algorithms can be proved formally by examining the system state sequences that can be generated by the algorithms in question.

The paper is organized as follows: In the remainder of this section we motivate our model by informally describing the problem of dbs reliability. The model itself is introduced in Section 2. In Section 3 we use the model to define algorithms and prove their reliability properties. Section 4 extends the model to describe *certain* aspects of reliability in distributed dbs. Section 5 comments on the model and suggests extensions.

1.2 The Problem of dbs Reliability

A database is a set of data items. For each item x there is a designated address in stable storage (e.g. disk) where a value for x is stored; x's designated address may change over time. The values stored at the designated addresses of all data items at time t comprise the materialized database at t [HR82].

To update x, a transaction provides the dbs with x's new value by placing it in a (volatile) main memory buffer. Later the dbs may copy the

value of x from the buffer to x's designated address either directly or after first copying it to another area of stable storage, called the audit trail. So, in effect, values of a data item may be found in three places: in a main memory buffer, in the audit trail, or in the item's designated address.

Until a transaction T executes its "Commit" operation, it is always possible for the dbs to abort T. This means that all updates of T to the database must be "undone" (by setting all updated items to the values they had prior to being changed by T). Also any transaction that read values produced by the aborted transaction T must be (recursively) aborted as well. Once, however, the dbs agrees to process a transaction's commit operation, it cannot subsequently abort it.

The problem of database reliability is essentially a fault tolerance problem. We want the dbs to correctly process database operations submitted by transactions, in spite of transaction and system failures. A transaction failure amounts to that transaction's being aborted. A system failure amounts to the loss of volatile storage: in such an event, the dbs must reconstruct from the contents of stable storage only (i.e. the materialized database and the audit trail), the state of the database reflecting the execution of exactly those transactions that were committed by the time the system failure occurred.

An algorithm for processing database operations is fault tolerant with respect to transaction and system failures if it can, at all times, (a) abort an uncommitted transaction without having to (recursively) abort a committed one and (b) reconstruct the correct state of every data item from values stored in stable storage.

Accordingly we can distinguish two aspects of dbs reliability. The first aspect concerns the parallel processing of transactions. We want to allow only such parallel executions as will not require aborting a committed transaction in the face of a transaction or system failure. In [H82] we characterized executions that have this property, called recoverable executions. In such executions, a transaction reads the value of a data item only if it was written by a committed transaction.

The second aspect of dbs reliability arises from the general principle that fault tolerance requires redundancy. All known dbs reliability algorithms require that some "redundant" data be kept in stable storage. This data is redundant in that it is only needed in case of a (transaction or system) failure. In essence, this redundant data forms what we previously called the audit trail.

Bernstein and others [BGH82] present a classification of algorithms in terms of their "undo/redo characteristics". We say that an algorithm may require undo if it allows an uncommitted transaction to record updates in the materialized database; if the uncommitted transaction aborts, its update must be undone. An algorithm may require redo if it allows a transaction to commit before allits updates have been recorded in the materialized database; if the system fails, the transaction's updates must be redone. We thus have four "classes" of algorithms:

Requirements	
UNDO	REDO
maybe	maybe
maybe	never
never	maybe
never	never
	UNDO maybe maybe never

In Section 3 we'll cast these four classes of algorithms in terms of our model.

2. THE MODEL

2.1 Preliminaries

In this subsection we summarize some basic definitions and notation. For motivation and fuller discussion see [H82].

Let $D = \{x,y,z,...\}$ be a set of data items. The symbols $R_i[x]$, $W_i[x]$, R_i , C_i where $i \in IN$ and $x \in D$ are called database operations. $R_i[x]$ is the read operation (issued by transaction i for data item x). $W_i[x]$ is the write operation (issued by transaction i for data item x). R_i is the abort operation of transaction i and C_i is the commit operation of transaction i.

Two operations conflict iff

- (a) they are read or write operations accessing the same data item and (at least) one of them is a write operation; or
- (b) (at least) one of them is a commit or an abort operation.

A transaction T_i , is a partial order $(OP_i, <_i)$, where

- (i) $OP_i \subseteq \{R_i[x], W_i[x], A_i, C_i | x \in D\}$
- (ii) Any two conflicting operations in OP_i are ordered by $<_i$,
- (iii) $A_i \in OP_i$ iff $C_i \notin OP_i$, and
- (iv) If $A_i \in OP_i$ then for every $a \in OP_i \{A_i\}$, $a < A_i$ and if $C_i \in OP_i$ then for every $a \in OP_i \{C_i\}$, $a < C_i$.

A complete log L (over transactions T_i $1 \le i \le n$), is a partial order,

 $L = (OP_L, <_L)$ where

- (i) $OP_L = U OP_i$, i=1
- (ii) $<_L \ge <_i$, for $1 \le i \le n$, and
- (iii) Any two conflicting operations in OP_L are ordered by \leq_L .

A $log \ L = (OP_L, <_L)$ is a prefix of some complete $log \ L' = (OP_L, <_L)$. If $C_i \in OP_L$, transaction T_i is committed in L; if $A_i \in OP_L$, T_i is aborted in L. $Com(L) \stackrel{\Delta}{=} \{T_i \mid C_i \in OP_L\}$. The projection of a $log \ L$ onto a set of transactions T, denoted $T_T(L)$, is the restriction of L to the domain $OP_L = U = OP_i$.

For mathematical simplicity we expand logs with a (fictitious) initializing transaction T_0 that writes all data items and commits [P79]. That is, $T_0 = (OP_0, <_0)$, where $OP_0 = \{W_0[x] \mid x \in D\} \cup \{C_0\}$, and $<_0 = \{(W_0[x], C_0) \mid x \in D\}$. All of T_0 's operations precede all other operations in the log; i.e. if the log is L, for any $a \in OP_0$ and any $b \in OP_1 - OP_0$, $a <_1 b$.

The Herbrand semantics of read and write operations in a log L, denoted $M_L(R_i[x])$ and $M_L(W_i[x])$ respectively, are defined inductively on $<_L$ as follows: ‡

- (1) $M_L(R_i[x]) = M_L(W_j[x])$, where $W_j[x] <_L R_i[x]$ and there is no $W_k[x] \in OP_L$ such that $W_j[x] <_L W_k[x] <_L R_i[x]$.
- (2) Let $R_i[y_t]$ $1 \le t \le m$ be all the read operations in T_i such that $R_i[y_t] \le W_i[x]$. Then $M_L(W_i[x]) = \hat{g}_{ix}(M_L(R_i[y_1]), \dots, M_L(R_i[y_m]))$. In particular, if there is no $R_i[y] \in OP_i$ such that $R_i[y] \le W_i[x]$, then $M_L(W_i[x]) = \hat{g}_{ix}()$.

 $^{^{\}dagger}P = (X,<)$ is a prefix of P' = (X',<') if $X \subseteq X'$, < is the restriction of <' to X (i.e., a < b iff a,b \in X and a <' b), and X is closed under <' (i.e., if a \in X and for some b \in X', b <' a then b \in X).

L being a discrete (indeed, finite) partial order, we may induce on the order of its elements. (The "order" of an element of a partial order <, is equal to the number of elements that precede it in <.)

 g_{ix} is an uninterpreted function symbol.

Note that, by our assumption concerning the expansion of logs by the initializing transaction T_0 , part (1) of the definition is well-defined, since $W_0[x]$ precedes any $R_1[x]$. Also, by part (2) of the definition, $M_L(W_0[x]) = \hat{g}_{ox}(\cdot)$, for any $x \in D$.

The Herbrand universe of a log L is the set $\operatorname{HU}(L) = \{\operatorname{M}_L(a) \mid a \in \operatorname{OP}_L \text{ is a read or write operation}\}$. The semantics of a log L, is the function $\operatorname{M}[L]: D \to \operatorname{HU}(L)$ defined by $\operatorname{M}[L](x) = \operatorname{M}_L(W_1[x])$ where $\operatorname{W}_1[x]$ is the L -maximum write operation on x in OP_L . A log L is recoverable iff for every prefix L' of L and for all $a \in \operatorname{OP}_T$ (L'), where a is a read or write, $\operatorname{M}_{L'}(a) = \operatorname{M}_T$ (L') (a). This implies that at no point in the execution represented by log L has a committed transaction read a value written by an uncommitted transaction [H82]. The class of recoverable logs is denoted RC.

2.2 Basic Definitions

Let T be the set of transactions with operations accessing the database D. Let \mathscr{Z} be the set of logs over transactions in T. Let $HU = \bigcup HU(L)$ be the Herbrand universe of logs in \mathscr{Z} .

A database state, S, is a mapping from D to HU; S: D \rightarrow HU. A system state, σ , is a triple $\sigma = (S_{\sigma}, SWI_{\sigma}, L_{\sigma})$ where S_{σ} is a database state, $L_{\sigma} \in \mathscr{L}$ and the stable write information at state σ , $SWI_{\sigma} \subseteq D \times HU \times T$.

In terms of the informal discussion in Section 1.2, the intended interpretation of the system state σ is as follows. The database state represents the materialized database: $S_{\sigma}(x) = v$, means that the value of x stored in the materialized database is v. The stable write information, SWI_{σ} , represents the audit trail: $(x,v,T_i) \in SWI_{\sigma}$, means that transaction T_i wrote value v into data item x, and that fact is recorded in the audit trail.

Finally, L_{σ} represents the execution that produced the current system state, σ : if $W_{\mathbf{i}}[\mathbf{x}] \in \mathsf{OP}_{\mathbf{L}_{\sigma}}$, the value $M_{\mathbf{L}_{\sigma}}(W_{\mathbf{i}}[\mathbf{x}])$ is written to a volatile memory buffer. When the dbs records this value in the audit trail, the new system state, say σ' , will have $(\mathbf{x}, M_{\mathbf{L}_{\sigma}}(W_{\mathbf{i}}[\mathbf{x}]), T_{\mathbf{i}}) \in \mathsf{SWI}_{\sigma'}$. Similarly, when the value is recorded in the materialized database, the new system state, say σ'' , will have $S_{\sigma''}(\mathbf{x}) = M_{\mathbf{L}_{\sigma}}(W_{\mathbf{i}}[\mathbf{x}])$. Hoping that this informal explanation provides a basis for understanding the basic building blocks of our model we revert to its further formal development.

Definition 2.1. The last committed writer of (data item) \times in (system state) σ , $\text{lcw}_{\sigma}(x)$, is the transaction T_i such that C_i , $W_i[x] \in \text{OP}_{L_{\sigma}}$ and for all other T_j such that C_j , $W_j[x] \in \text{OP}_{L_{\sigma}}$ we have $W_j[x] <_{L_{\sigma}} W_i[x]$. $\square_{2.1}$

Note that by the assumption concerning the initial transaction T_0 , and the fact that write operations on the same data item (being conflicting operations) must be related in the partial order of a log, $lcw_0(x)$ is always well-defined.

Definition 2.2. The committed database state of (system state) σ , CS_{σ} , is the database state defined by: $CS_{\sigma}(x) \stackrel{\Delta}{=} M_{L_{\sigma}}(W_{\mathbf{i}}[x])$, where $T_{\mathbf{i}} = lcw_{\sigma}(x)$. $\square_{2.2}$

Let Σ be the set of all system states. We distinguish a special system state, σ_0 , called the *initial system state* where: $S_{\sigma_0}(x) = \hat{g}_{0x}(\cdot)$, for all $x \in D$, $SWI_{\sigma_0} = \emptyset$ and $L_{\sigma_0} = \varepsilon$.

<u>Definition 2.3</u>. A system state σ is resilient iff for all $x \in D$, either (R1) $S_{\sigma}(x) = CS_{\sigma}(x)$, or

We use the symbol ϵ for the "empty log", containing only T_0 ; i.e. $OP_{\epsilon} = OP_0$ and $c_{\epsilon} = c_0$.

(R2)
$$(x,CS_{\sigma}(x),1cw_{\sigma}(x)) \in SWI$$
.

D_{2.3}

Intuitively, the definition of resiliency says that the "correct" value of every data item in the system state described by σ can be found in stable storage—either in the "materialized database" (S_{σ}), or in the "audit trail" (SWI_{σ}).

If (R1) or (R2) holds for a particular x in a system state σ we'll say that σ is resilient for x.

LEMMA 2.4. σ_0 is resilient.

Proof. For all $x \in D$, $CS_{\sigma_0}(x) \stackrel{\triangle}{=} M_{\varepsilon}(W_0[x]) \stackrel{\triangle}{=} \hat{x} \stackrel{\triangle}{=} S_{\sigma_0}(x)$. Hence (R1) holds for all $x \in D$ for σ_0 .

2.2 Algorithms and their Properties

An algorithm $\mu_{\mathbf{A}}$ (for processing data operations) is a set-valued operator on Σ , $\mu_{\mathbf{A}} \colon \Sigma \to 2^{\Sigma}$, such that if $\sigma' \in \mu_{\mathbf{A}}(\sigma)$ then \mathbf{L}_{σ} , $\supseteq \mathbf{L}_{\sigma}$.

A history α is a finite sequence of system states; i.e. $\alpha \in \Sigma^*$. A history $\sigma_1 \sigma_2 \dots \sigma_n$ is μ_A -compatible iff $\sigma_{i+1} \in \mu_A(\sigma_i)$ for $1 \le i \le n$.

Definition 2.5. An algorithm μ_A is correct iff for any μ_A -compatible history $\sigma_0\sigma_1\dots\sigma_n$, σ_n is resilient.

Intuitively, an algorithm is correct if it transforms the system state in such a way that at any state reachable form the initial one, we can construct the "correct" value of every data item from information in stable storage.

Here and throughout the paper, by σ_0 we mean specifically the initial system state, whereas by σ_i $1 \le i \le n$ we mean arbitrary elements of Σ .

It goes without saying that there is quite a gap between this description of an algorithm and its concrete description in terms of a realistic programming language. The assumption underlying our model is that each transition from σ to each element of $\mu_{A}(\sigma)$ can be implemented atomically. If this is not the case, our results are mathematically sound but pragmatically meaningless. Often such an atomic implementation is far from obvious. For implementation-related issues of the algorithms to be described later see the references cited in Section 1.2.

Definition 2.6. An algorithm $\mu_{\mathbf{A}}$ may require redo iff there exists a $\mu_{\mathbf{A}}$ -compatible history $\sigma_0 \sigma_1 \dots \sigma_n$ s.t.

$$\exists \mathbf{T}_{\mathbf{i}} \exists \mathbf{x} [\mathbf{S}_{\sigma_{\mathbf{n}}}(\mathbf{x}) = \mathbf{M}_{\mathbf{L}_{\sigma_{\mathbf{n}}}}(\mathbf{W}_{\mathbf{i}}[\mathbf{x}]) \land \mathbf{C}_{\mathbf{i}} \in \mathsf{OP}_{\mathbf{L}_{\sigma_{\mathbf{n}}}} \land \mathbf{T}_{\mathbf{i}} \neq \mathsf{lcw}_{\sigma_{\mathbf{n}}}(\mathbf{x})] . \qquad \Box_{2.6}$$

This definition says that starting in the initial system state the algorithm has produced a system state in which the value recorded in the "materialized database" for an item x was written by a committed transaction—but not the *last* committed writer of x.

<u>Definition 2.7.</u> An algorithm μ_A may require undo iff there is a μ_A -compatible history $\sigma_0\sigma_1...\sigma_n$ such that

$$\exists x [S_{\sigma_n}(x) = M_{L_{\sigma_n}}(W_i[x]) \land C_i \notin OP_{L_{\sigma_n}}] \qquad .$$

This definition says that starting from the initial system state, the algorithm has produced a system state in which the value of some data item x recorded in the "materialized database" was produced by an uncommitted transaction.

2.3 Well-formed Algorithms

In this subsection we develop the notion of a "well-formed algorithm" (wfa). Informally, an algorithm is well-formed if it can, at any time, respond to "correctly" submitted user (transaction) operations. This means, for instance, that if an algorithm has produced a system state σ , it must be able at the next transition to move to a system state σ , reflecting that an update of a non-terminated transaction was received by the algorithm. This could be formally stated as follows in the "language" of our model:

$$\forall \mathtt{T}_{\mathtt{i}} \left[\mathtt{C}_{\mathtt{i}}, \mathtt{A}_{\mathtt{i}} \not\in \mathtt{OP}_{\mathtt{L}_{\sigma}} \Rightarrow \exists \sigma' \in \mu_{\mathtt{A}}(\sigma) \left[\mathtt{L}_{\sigma'} = \mathtt{L}_{\sigma} \circ W_{\mathtt{i}} \left[\mathtt{x} \right] \right] \right]^{\frac{1}{4}} \ .$$

Note that our definition of "correct algorithm" made no provision for this. So, for example, the algorithm, μ_{silly} , defined by: $\forall \sigma \in \Sigma$ $\mu_{\text{silly}}(\sigma) = \sigma_0$ is correct according to Definition 2.5. (since the only μ_{silly} -compatible histories starting with σ_0 are of the form $\sigma_0\sigma_0...\sigma_0$, and σ_0 is resilient by Lemma 2.4). So, sensible algorithms must not only be correct but well-formed as well.

Definition 2.8. An algorithm μ_{A} is well-formed iff it satisfies both of the following properties:

<u>WFAl</u>: For any μ_A -compatible history $\sigma_0 \sigma_1 \dots \sigma_n$, there is some $\sigma \in \mu_A(\sigma_n)$ where $L_{\sigma} = L_{\sigma} \circ a_i$ for any T_i s.t. C_i , A_i , $a_i \notin OP_{L_{\sigma}}$, $a_i \in \{R_i[x], W_i[x], A_i\}$.

That is, a transaction not yet committed or aborted.

The notation " L_{σ} , = L_{σ} a " means that L_{σ} , is some (arbitrary) extension of L_{σ} with a tacked at the end; formally, $OP_{L_{\sigma}} = OP_{L_{\sigma}} \cup \{a_i\}$ and there is no $b_j \in \bar{OP}_{L_{\sigma}}$, s.t. $a_i <_{L_{\sigma}}, b_j$.

Recall that, by definition, there can be at most one read and one write operation for a given data item in any given transaction. This restriction is only made for the sake of keeping the notation simpler: the results in this paper in no way depend on this assumption.

<u>WFA2:</u> For any μ_A -compatible history $\alpha = \sigma_0 \sigma_1 \dots \sigma_n$ there is a history $\beta = \tau_1 \tau_2 \dots \tau_m$ such that

- (a) $\alpha\beta$ is $\mu_{\mbox{\scriptsize A}}\mbox{-compatible, and}$
- (b) $L_{\tau_m} = L_{\sigma_n} \circ C_i$, where C_i , $A_i \notin OP_{L_{\sigma_n}}$.

Informally, WFAl says that a wfa can at any time receive and immediately start processing any read, write or abort operation for a non-terminated transaction. WFA2 is a little different because a transaction cannot necessarily be committed immediately after a commit request is made. The algorithm for processing transaction operations may have some "house cleaning" to do before it can commit a transaction (e.g., transfer some values from "volatile buffers" to the "materialized database" or the "audit trail"). However, we do require that a wfa be able to commit a (non-terminated) transaction with finite delay from any system state it has reached (starting from σ_0).

<u>Definition 2.9</u>. The set of histories associated with a log L, relative to algorithm $\mu_{\rm h}$, $h_{\rm h}$ (L), is defined as:

$$h_{\mathbf{A}}(\mathbf{L}) = \{\alpha \mid \alpha = \sigma_0 \sigma_1 \dots \sigma_n \text{ is } \mu_{\mathbf{A}} \text{-compatible and } \mathbf{L}_{\sigma_n} = \mathbf{L}\}.$$

Informally, $h_{\bf A}({\bf L})$ is the set of histories that could be generated by algorithm $\mu_{\bf k}$ in response to the execution represented by L.

2.4 Properties of wfa's

The next theorem shows that wfa's never "get stuck" in processing operations--i.e., if any (legal) operation is submitted at any time, a wfa can transform the current system state to one that reflects the fact that it

received and started processing the operation. Recall from the preceding section that this was the motivation for introducing wfa's.

THEOREM 2.10. If μ_A is well-formed then for any $\log L$, $h_A(L) \neq \emptyset$

Proof. By induction on L.

Basis. L $\equiv \epsilon$ (the empty log). Since σ_0 is μ_A -compatible (in fact for any μ_A --not just well-formed μ_A !) and L $\sigma_0 = \epsilon$ by definition of σ_0 , we have that $\sigma_0 \in h_A(\epsilon)$. Hence $h_A(L) \neq \emptyset$.

Induction Step. Suppose this is true for a log L'. We'll show that it holds for $L = L' \circ a_i$, where $a_i \in \{R_i[x], W_i[x], C_i, A_i\}$.

For $\mathbf{a_i} \equiv \mathbf{R_i}[\mathbf{x}]$ ($\mathbf{W_i}[\mathbf{x}]$, $\mathbf{A_i}$, respectively) consider (by induction hypothesis) any $\sigma_0 \sigma_1 \dots \sigma_n \in \mathbf{h_A}(\mathbf{L'})$ and let $\sigma_{n+1} \in \mu_{\mathbf{A}}(\sigma_n)$ such that $\mathbf{L_{\sigma_{n+1}}} = \mathbf{L_{\sigma_{n}}} \circ \mathbf{R_i}[\mathbf{x}]$ ($\mathbf{L_{\sigma_{n}}} \circ \mathbf{W_i}[\mathbf{x}]$, $\mathbf{L_{\sigma_{n}}} \circ \mathbf{A_i}$, respectively). σ_{n+1} exists by WFA1. Note that $\mathbf{L_{\sigma_{n}}} = \mathbf{L'}$ and hence $\mathbf{L_{\sigma_{n+1}}} = \mathbf{L'} \circ \mathbf{R_i}[\mathbf{x}]$ ($\mathbf{L'} \circ \mathbf{W_i}[\mathbf{x}]$, $\mathbf{L'} \circ \mathbf{A_i}$, respectively). So, $\mathbf{L_{\sigma_{n+1}}} = \mathbf{L}$. Thus, $\sigma_0 \sigma_1 \dots \sigma_{n+1} \in \mathbf{h_A}(\mathbf{L})$.

For the remaining case, $a_i \equiv C_i$, consider any $\alpha = \sigma_0 \sigma_1 \dots \sigma_n \in h_A(L^i)$ and let $\beta = \tau_1 \tau_2 \dots \tau_m$ be such that $\alpha\beta$ is μ_A -compatible and $L_{\tau_m} = L_{\sigma_n} \circ C_i$ β exists by WFA2. Note that $L' = L_{\sigma_n}$ (by inductive hypothesis) and hence C_i , $A_i \not\in \text{OP}_{L^i}$ (otherwise, $L = L' \circ C_i$ would not be a log). Since $L_{\tau_m} = L_{\sigma_n} \circ C_i$ and $L_{\sigma_n} = L'$ we have $L_{\tau_m} = L' \circ C_i = L$. Therefore $\alpha\beta \in h_A(L)$ and hence $h_A(L) \neq \emptyset$.

LEMMA 2.11. Let μ_A be a wfa and L be any log. Then if $\sigma_0 \sigma_1 \dots \sigma_n \in h_A(L)$ and $\sigma_0 \sigma_1 \dots \sigma_m \in h_A(L)$, $CS_{\sigma_n} = CS_{\sigma_m}$.

<u>Proof.</u> Immediate from the fact that $L_{0} = L_{\tau} = L$.

LEMMA 2.12. If $L_{\sigma} \in RC$ (i.e., L_{σ} is a recoverable log) then $CS_{\sigma} = M[\pi_{Com(L_{\sigma})}(L_{\sigma})].$

Proof. By definition, for any $x \in D$,

$$CS_{\sigma}(x) = M_{L_{\sigma}}(W_{i}[x]), \text{ where } T_{i} = 1cW_{\sigma}(x).$$
 (2.12.1)

That is, $W_i[x]$, $C_i \in OP_{L_{\overline{O}}}$ and for any $j \neq i$ such that C_j , $W_j[x] \in OP_{L_{\overline{O}}}$, $W_j[x] <_{L_{\overline{O}}} W_i[x]$. By definition of log projection, it is easy to see that $W_i[x]$, $C_i \in OP_{\pi_{Com}(L_{\overline{O}})}$ and that if $W_j[x]$, $C_j \in OP_{\pi_{Com}(L_{\overline{O}})}$ for some $j \neq i$, then $W_j[x] <_{\pi_{Com}(L_{\overline{O}})} (L_{\overline{O}})$ $W_i[x]$. Hence,

$$M[\pi_{Com(L_{\sigma})}^{(L_{\sigma})}](x) = M_{\pi_{Com(L_{\sigma})}^{(L_{\sigma})}}^{(L_{\sigma})}(x) . \qquad (2.12.2)$$

By definition of recoverability, we have

$$^{M}_{\pi_{\text{Com}(L_{\sigma})}(L_{\sigma})}(L_{\sigma})^{(W_{i}[x])} = ^{M}_{L_{\sigma}}(W_{i}[x])$$
 (2.12.3)

From (2.12.1), (2.12.2), (2.12.3) we get

$$CS_{\sigma}(x) = M[\pi_{Com(L_{\sigma})}(L_{\sigma})](x)$$
, for any $x \in D$.

The next theorem relates wfa's to recoverable executions.

THEOREM 2.13. If $\mu_{\mathbf{A}}$ is a wfa, $\mathbf{L} \in \mathbb{RC}$, $\sigma_0 \sigma_1 \dots \sigma_n \in \mathbf{h}_{\mathbf{A}}(\mathbf{L})$, $\sigma_0 \sigma_1 \dots \sigma_n \in \mathbf{h}_{\mathbf{A}}(\mathbf{L})$, then $\mathbf{CS}_{\sigma_n} = \mathbf{CS}_{\tau_n}$.

Informally this theorem says that the committed database state of any system state reached by a wfa in response to the operations of a recoverable execution E is the same as the committed database state of a system state reached by the same wfa in response to execution E! which is the same as

E except that the operations of aborted or uncommitted transactions of E never happened. Note that the committed database state at any time t is what we want to reconstruct as the "correct" database state, if a system failure happens at t.

Now, if the algorithm, in addition to being well-formed, is also correct, we have enough information in "stable storage" to reconstruct that database state.

In the following section we shall study four algorithms (the four classes mentioned on Section 1.2) and shall prove that they are correct and well-formed. From Theorem 2.13 and these anticipated facts we conclude that any of these four algorithms, coupled with a scheduler that produces recoverable logs, provides tolerance to the faults under consideration—that is, transaction and system failures.

THE FOUR BASIC ALGORITHMS

3.1 Introduction

In this section we examine in detail the four algorithms introduced in Section 1.2. First we define these algorithms in the style of the previous section; that is, as (non-deterministic) state transition functions. We define four such functions $\mu_{\mathbf{I}}$, $\mu_{\mathbf{III}}$, $\mu_{\mathbf{III}}$ and $\mu_{\mathbf{IV}}$ corresponding, respectively, to the four algorithms. The functions are defined by listing "types" of transitions as in [Ly82]. Each transition type is characterized by (a) a set of conditions on states and (b) state transformation rules, describing how the new state is to be obtained from the old. The idea is that if a state σ satisfies the conditions of a transition type of algorithm $\mu_{\mathbf{A}}$, and σ' is obtained from σ as described by the state transformation rules of that transition type, then $\sigma' \in \mu_{\mathbf{A}}(\sigma)$.

For every one of these four algorithms (state transition functions) we prove rigorously its correctness, good formation and properties with respect to redo and undo.

3.2 Algorithm I (both undo and redo)

ALGORITHM 3.1. $\sigma' \in \mu_{\tilde{I}}(\sigma)$ iff one of the following is the case:

I.l: [Submit a read operation]

Conditions:

$$\exists T_i \exists x [C_i, A_i, R_i [x] \notin OP_L]$$

Transformation rules:

$$S_{\sigma}^{\prime} = S_{\sigma}^{\prime}$$

$$SWI_{\sigma}^{\prime} = SWI_{\sigma}^{\prime}$$

$$L_{\sigma}^{\prime} = L_{\sigma} \circ R_{i}^{\prime}[x] \dagger$$

I.2: [Submit a write operation]

Conditions:

$$\exists T_i \exists x [C_i, A_i, W_i[x] \notin OP_{L_G}]$$

Transformation rules:

$$S_{\sigma}^{\dagger} = S_{\sigma}^{\dagger}$$
 $SWI_{\sigma}^{\dagger} = SWI_{\sigma}^{\dagger}$
 $L_{\sigma}^{\dagger} = L_{\sigma}^{\bullet}W_{i}^{\dagger}[x]$

I.3: [Record an update in the "audit trail"]

Conditions:

$$\exists \mathtt{T_i} \exists \mathtt{x} [\mathtt{C_i}, \mathtt{A_i} \not\in \mathtt{OP}_{\mathtt{L}_{\overline{\mathcal{O}}}} \land \mathtt{W_i} [\mathtt{x}] \in \mathtt{OP}_{\mathtt{L}_{\overline{\mathcal{O}}}}]$$

Transformation rules:

$$S_{\sigma}$$
, = S_{σ}
 SWI_{σ} , = $SWI_{\sigma} \cup \{(x, M_{L_{\sigma}}(W_{i}[x]), T_{i})\}$
 L_{σ} , = L_{σ}

I.4: [Record an update in the "materialized database"]

Conditions:

$$\exists \mathtt{T_i} \exists \mathtt{x!} (\mathtt{x}, \mathtt{M_{L_\sigma}} (\mathtt{W_i}[\mathtt{x}]), \mathtt{T_i}) \in \mathtt{SWI}_\sigma]$$

We adopt the convention that the existentially quantified objects in the conditions bound any free objects in the transformation rules. Hence the subscript i and the x in $R_i[x]$ here, refer to the transaction T_i and the data item x described by the conditions.

Transformation rules:

$$S_{\sigma}, (y) = \begin{cases} S_{\sigma}(y) & \text{if } y \neq x \\ M_{L_{\sigma}}(W_{i}[x]) & \text{if } y = x \end{cases}$$

$$SWI_{\sigma}, = SWI_{\sigma}$$

$$L_{\sigma}, = L_{\sigma}$$

I.5: [Commit a transaction]

Conditions:

$$\exists \mathtt{T_{i}} [\mathtt{C_{i}}, \mathtt{A_{i}} \not\in \mathtt{OP}_{\mathtt{L_{o}}} \land \forall \mathtt{W_{i}} [\mathtt{x}] \in \mathtt{OP}_{\mathtt{L_{o}}} [(\mathtt{x}, \mathtt{M_{L}} \ (\mathtt{W_{i}} [\mathtt{x}]), \mathtt{T_{i}}) \in \mathtt{SWI_{o}}]]$$

Transformation rules:

$$S_{\sigma}$$
, = S_{σ}
 SWI_{σ} , = SWI_{σ}
 L_{σ} , = $L_{\sigma} \circ C_{i}$

I.6: [Abort a transaction]

Conditions

$$\exists T_i [C_i, A_i \notin OP_{L_G}]$$

Transformation rules

$$S_{\sigma'} = S_{\sigma}$$

$$SWI_{\sigma'} = SWI_{\sigma}$$

$$L_{\sigma'} = L_{\sigma} \cdot A_{i}$$

1.7: [Restore update of aborted transaction in "materialized database"]

Conditions:

$$\exists T_i \exists x [A_i \in OP_{L_{\sigma}} \land S_{\sigma}(x) = M_{L_{\sigma}}(W_i[x])]$$

Transformulation rules:

$$S_{\sigma}^{},(y) = \begin{cases} S_{\sigma}^{}(y) & \text{if } y \neq x \\ \\ CS_{\sigma}^{}(x) & \text{if } y = x \end{cases}$$

$$SWI_{\sigma}$$
, = SWI_{σ}
 L_{σ} , = L_{σ}

1.8: [Discard unnecessary information from the "audit trail"]

Conditions:

$$\exists \mathtt{T}_{\mathbf{i}} \exists \mathtt{x} [[\mathtt{C}_{\mathbf{i}} \in \mathtt{OP}_{\mathtt{L}_{\mathcal{O}}} \land \mathtt{T}_{\mathbf{i}} \neq \mathtt{lcw}_{\mathcal{O}}(\mathtt{x})] \lor \mathtt{A}_{\mathbf{i}} \in \mathtt{OP}_{\mathtt{L}_{\mathcal{O}}}]$$
Transformation rules:

$$S_{\sigma}$$
, = S_{σ}
 SWI_{σ} , = $SWI_{\sigma} - \{(x, M_{L_{\sigma}}(W_{i}[x]), T_{i})\}$
 L_{σ} , = L_{σ} .

⁰3.1

THEOREM 3.2. Algorithm μ_{I} is correct.

<u>Proof.</u> For any $\mu_{\underline{I}}$ -compatible history $\sigma_0\sigma_1...\sigma_n$, we'll show, by induction, that σ_k is resilient, $0 \le k \le n$.

Basis: σ_0 is resilient by Lemma 2.4.

Induction Step: Suppose σ_k is resilient for some k, $0 \le k \le n$. By μ_1 -compatibility of $\sigma_0 \sigma_1 \dots \sigma_n$ we have that $\sigma_{k+1} \in \mu_1(\sigma_k)$. Consider each transition type.

Types I.1-3 and 6: In all these cases, $lcw_{\sigma_{k+1}} = lcw_{\sigma_k}$, $CS_{\sigma_{k+1}} = CS_{\sigma_k}$, $S_{\sigma_{k+1}} = S_{\sigma_k}$ and $SWI_{\sigma_{k+1}} \supseteq SWI_{\sigma_k}$. Thus if (R1) or (R2) holds for any $x \in D$ in σ_k it will hold for x in σ_{k+1} . By inductive hypothesis then, σ_{k+1} is resilient.

Type I.4: Let T_i , x be such that $S_{G_{k+1}}(x) = M_{L_G}(W_i[x]) \neq S_{G_k}(x)$. Since $1_{CW_{G_{k+1}}}(x) = 1_{CW_{G_{k+1}}}(x) = C_{G_k}(x) = C_{G_k}(x$

$$T_{\ell} = lcw_{\sigma_{\mathbf{r}}}(\mathbf{x}) \qquad \text{for } p \leq \mathbf{r} \leq k+1. \tag{3.2.1}$$

By the I.5 conditions, we have that $(x,M_L,(W_{\ell}[x]),T_{\ell}) \in SWI_{\sigma}$. Clearly, σ_{p-1} $M_{L_{\sigma}}(W_{\ell}[x]) = M_{L_{\sigma}}(W_{\ell}[x]) \text{ for } p \leq r \leq k+1 \text{ since } L_{\sigma} \text{ is a prefix of } p-1$ all L_{σ} . We claim that

$$(x,M_{L_{\sigma_r}}(W_{\ell}[x]),T_{\ell}) \in SWI_{\sigma_r} \text{ for } p-1 \leq r \leq k+1$$
 (3.2.2)

For, suppose not. Then for some q, $p < q \le k$, $(x, M_{L_{\sigma_{q-1}}} (W_{\ell}[x]), T_{\ell}) \in SWI_{\sigma_{q-1}}$ SWI_Q. By inspection of μ_{I} , the transition from σ_{q-1} to σ_{q} can only be of type I.8 (because that is the only type in which elements of SWI are deleted). Since $A_{i} \notin OP_{L_{\sigma_{q-1}}}$ we must have, by the I.8 conditions, that $T_{\ell} \ne 1cw_{\sigma_{q-1}}$ (x) but this contradicts (3.2.1), as $p \le q-1 \le k$. Therefore, q-1 (3.2.2) is true and in particular, since $T_{\ell} = 1cw_{\sigma_{k+1}}$ (x) and thus $CS_{\sigma_{k+1}} (x) = M_{L_{\sigma_{k+1}}} (w_{\ell}[x])$, we get that $(x, CS_{\sigma_{k+1}} (x), 1cw_{\sigma_{k+1}} (x)) \in SWI_{\sigma_{k+1}}$. Hence, (\tilde{R}^2) holds for x in σ_{k+1} , as needed.

Type I.7: Let T_i , x be such that $A_i \in OP_L$ and $S_{O_{k+1}}(x) = CS_{O_k}(x)$. For any $y \in D - \{x\}$, if (R1) or (R2) holds for y^k in O_k it also does in O_{k+1} . by inductive hypothesis, then, O_{k+1} is resilient for $y \in D - \{x\}$. Since $CS_{O_{k+1}} = CS_{O_k}$ we also have that $S_{O_{k+1}}(x) = CS_{O_{k+1}}(x)$ and thus (R1) holds for x in O_{k+1} . So O_{k+1} is resilient.

Type I.8: Let T_i , x be such that $(x, M_{L_{\sigma_k}}(W_i[x]), T_i) \in SWI_{\sigma_k} - SWI_{\sigma_{k+1}}$. Since $lcw_{\sigma_{k+1}} = lcw_{\sigma_k}$, $CS_{\sigma_{k+1}} = CS_{\sigma_k}$ and $S_{\sigma_i} = S_{\sigma_i}$, if (R1) holds for any $y \in D$ in σ_k it also holds in σ_{k+1} . Also, if (R2) holds for $y \in D - \{x\}$ in σ_k it holds in σ_{k+1} , as well. So, let's examine the remaining case, namely when (R2) holds for x in σ_k . We then have $(x, CS_{\sigma_k}(x), CS_{\sigma_k}(x), CS_{\sigma_k}(x)) \in SWI_{\sigma_k} = SWI_{\sigma_k}$, only if $cw_{\sigma_k}(x) = T_i$. By the I.8 conditions, however, either $cw_{\sigma_k}(x) = T_i$ in which $cw_{\sigma_k}(x) = T_i$. By the I.8 conditions, however, either $cw_{\sigma_k}(x) = T_i$ and $cw_{\sigma_k}(x) = T_i$ and cw

This concludes the induction step and thereby the proof.

⁰3.2

THEOREM 3.3. Algorithm μ_{T} may require both redo and undo.

<u>Proof.</u> Consider the history $\sigma_0^{\sigma_1\sigma_2\sigma_3\sigma_4\sigma_5\sigma_6}$ defined by: $(\sigma_0^{\sigma_1})$ is the initial system state)

$$s_{\sigma_1} = s_{\sigma_0}$$
, $swi_{\sigma_1} = swi_{\sigma_0}$, $L_{\sigma_1} = L_{\sigma_0} \circ w_1[x]$

$$s_{\sigma_2} = s_{\sigma_0}$$
, $swi_{\sigma_2} = swi_{\sigma_1}$, $L_{\sigma_2} = L_{\sigma_1} \cdot w_2[y]$

$$s_{\sigma_3} = s_{\sigma_2}$$
, $swi_{\sigma_3} = swi_{\sigma_2} \cup \{(x, \hat{g}_{1x}(), T_1)\}$, $L_{\sigma_3} = L_{\sigma_2}$

$$s_{\sigma_4} = s_{\sigma_3}$$
, $swi_{\sigma_4} = swi_{\sigma_3} U\{(x, \hat{g}_{2y}(), T_2)\}$, $L_{\sigma_4} = L_{\sigma_3}$

$$S_{\sigma_{5}}(z) = \begin{cases} S_{\sigma_{4}}(z) & \text{if } z \neq x \\ & & \text{,} & \text{SWI}_{\sigma_{5}} = \text{SWI}_{\sigma_{4}}, \quad L_{\sigma_{5}} = L_{\sigma_{4}} \\ \hat{g}_{1x}(\cdot) & \text{if } z = x \end{cases}$$

$$S_{\sigma_6} = S_{\sigma_5}$$
, $SWI_{\sigma_6} = SWI_{\sigma_5}$, $L_{\sigma_6} = L_{\sigma_5} \circ C_2$

It can be readily verified that this is a $\mu_{\rm I}$ -compatible history ($\sigma_{\rm O}$ goes to $\sigma_{\rm I}$ and $\sigma_{\rm I}$ to $\sigma_{\rm 2}$ by I.2 transitions, $\sigma_{\rm 2}$ to $\sigma_{\rm 3}$ and $\sigma_{\rm 3}$ to $\sigma_{\rm 4}$ by I.3 transitions, $\sigma_{\rm 4}$ to $\sigma_{\rm 5}$ by an I.4, and $\sigma_{\rm 5}$ to $\sigma_{\rm 6}$ by an I.5 transition), that $S_{\sigma_{\rm 6}}(y) = \hat{g}_{\rm Oy}(\cdot) = M_{\rm L}_{\sigma_{\rm 6}}(w_{\rm O}[y])$, $C_{\rm O} \in {\rm OP}_{\rm L}_{\sigma_{\rm 6}}$ and $C_{\rm OP}(y) = C_{\rm OP}$

THEOREM 3.4. Algorithm μ_{I} is well-formed.

Proof. Satisfaction of WFAl by $\mu_{\mathbf{I}}$ follows immediately from transition types I.1,2 and 6. It remains to show that $\mu_{\mathbf{I}}$ satisfies WFA2. Let $\alpha = \sigma_0 \sigma_1 \dots \sigma_n$ be any $\mu_{\mathbf{I}}$ -compatible history. Let $T_{\mathbf{I}}$ be such that $C_{\mathbf{I}}$, $A_{\mathbf{I}} \notin \mathrm{OP}_{\mathbf{L}_{\sigma_n}}$.

Case 1: If for all $W_{\mathbf{I}}[\mathbf{x}] \in \mathrm{OP}_{\mathbf{L}_{\sigma_n}}$, $(\mathbf{x}, M_{\mathbf{L}_{\sigma_n}}(W_{\mathbf{I}}[\mathbf{x}]), T_{\mathbf{I}}) \in \mathrm{SWI}_{\sigma_n}$, the I.5 conditions are satisfied and we may define $\tau_{\mathbf{I}} \in \mu_{\mathbf{I}}(\sigma_n)$ such that $L_{T_{\mathbf{I}}} = L_{\sigma_n} \circ C_{\mathbf{I}}$. Since $\sigma_0 \sigma_1 \dots \sigma_n \tau_1$ is $\mu_{\mathbf{I}}$ -compatible, we have that $\mu_{\mathbf{I}}$ satisfies WFA2, in this case.

 $\begin{array}{lll} \underline{\operatorname{Case}\ 2} \colon & \operatorname{Suppose}\ \operatorname{there}\ \operatorname{is}\ \operatorname{some}\ \ \times\ \operatorname{such}\ \operatorname{that}\ \ \operatorname{W}_{\mathbf{i}}[x] \in \operatorname{OP}_{\mathbf{L}_{\mathcal{O}}},\ \operatorname{yet} \\ & (x,\operatorname{M}_{\mathbf{L}_{\mathcal{O}_{\mathbf{n}}}}(\operatorname{W}_{\mathbf{i}}[x]),\operatorname{T}_{\mathbf{i}}) \not\in \operatorname{SWI}_{\mathcal{O}_{\mathbf{n}}}. & \operatorname{Let}\ \ x_1,x_2,\ldots,x_m \ \ \operatorname{be}\ \operatorname{all}\ \operatorname{such}\ \ x's. \ \operatorname{Because} \\ & \operatorname{C}_{\mathbf{i}},\ \operatorname{A}_{\mathbf{i}} \not\in \operatorname{OP}_{\mathbf{L}_{\mathcal{O}}} & \operatorname{and}\ \ \operatorname{W}_{\mathbf{i}}[x_k] \in \operatorname{OP}_{\mathbf{L}_{\mathcal{O}}},\ \ \operatorname{the}\ \operatorname{conditions}\ \operatorname{for}\ \mathbf{I}.3\ \operatorname{transitions}\ \operatorname{are} \\ & \operatorname{satisfied}\ \operatorname{for}\ \ T_{\mathbf{i}},x_k & 1\leqslant k\leqslant m. \ \ \operatorname{We}\ \operatorname{may}\ \operatorname{then}\ \operatorname{define}\ \ \tau_k & \operatorname{for}\ \ 0\leqslant k\leqslant m \ \ \operatorname{as} \\ & \operatorname{follows}. \ \ \operatorname{Let} \end{array}$

$$\begin{array}{l} \tau_0 \equiv \sigma_n \;\; , \qquad \qquad \text{and} \\ \\ S_{\tau_{k+1}} = S_{\tau_k} \;\; , \\ \\ SWI_{\tau_{k+1}} = SWI_{\tau_k} \cup \{(x_k, M_{L_{\tau_k}}(W_i[x_k]), T_i)\} \\ \\ \\ L_{\tau_{k+1}} = L_{\tau_k} \;\; , \qquad \qquad \text{for} \quad 0 \leqslant k \leqslant m \quad . \end{array}$$

Note that $\tau_{k+1} \in \mu_{\mathbf{I}}(\tau_k) = 0 \leq k \leq m$, by I.3 transitions. Clearly, for all $\mathbf{W}_{\mathbf{I}}[\mathbf{x}] \in \mathrm{OP}_{\mathbf{L}_{\mathcal{O}_{\mathbf{I}}}}$, $(\mathbf{x}, \mathbf{M}_{\mathbf{L}_{\mathcal{O}_{\mathbf{I}}}}(\mathbf{W}_{\mathbf{I}}[\mathbf{x}]), \mathbf{T}_{\mathbf{I}}) \in \mathrm{SWI}_{\mathbf{T}_{\mathbf{M}}}$, and since $\mathbf{M}_{\mathbf{L}_{\mathcal{O}_{\mathbf{I}}}}(\mathbf{W}_{\mathbf{I}}[\mathbf{x}]) = \mathbf{M}_{\mathbf{L}_{\mathbf{T}_{\mathbf{I}}}}(\mathbf{W}_{\mathbf{I}}[\mathbf{x}])$, $\mathbf{C}_{\mathbf{I}}, \mathbf{A}_{\mathbf{I}} \notin \mathcal{OP}_{\mathbf{L}_{\mathbf{T}_{\mathbf{M}}}}$ the condition for I.5 transitions are satisfied at $\tau_{\mathbf{M}}$ and we may define $\tau_{\mathbf{M}+1} \in \mu_{\mathbf{I}}(\tau_{\mathbf{M}})$ such that $\mathbf{L}_{\mathbf{T}_{\mathbf{M}+1}} = \mathbf{L}_{\mathbf{T}_{\mathbf{M}}} \circ \mathbf{C}_{\mathbf{I}}$. Let $\beta = \tau_{\mathbf{I}} \tau_{\mathbf{2}} \cdots \tau_{\mathbf{M}+1}$. $\alpha\beta$ is clearly $\mu_{\mathbf{I}}$ -compatible and since $\mathbf{L}_{\tau_{\mathbf{M}}} = \mathbf{L}_{\mathbf{O}_{\mathbf{I}}}$, $\mathbf{L}_{\tau_{\mathbf{M}+1}} = \mathbf{L}_{\mathbf{O}_{\mathbf{I}}} \circ \mathbf{C}_{\mathbf{I}}$. Thus $\mu_{\mathbf{I}}$ satisfies WFA2 in this case, as well.

3.3 Algorithm II (undo but no redo)

ALGORITHM 3.5. $\sigma' \in \mu_{II}(\sigma)$ iff one of the following is the case.

II.1: [Submit a read operation]

Conditions:

$$\exists T_i \exists x [C_i, A_i, R_i [x] \notin OP_{L_{Cl}}]$$

Transformation rules:

$$s_{\sigma}$$
, = s_{σ}

$$L_{\sigma}^{\prime} = L_{\sigma}^{\circ}R_{i}[x]$$

II.2: [Submit a write operation]

Conditions:

$$\exists T_i \exists x [C_i, A_i, W_i[x] \notin OP_{L_{\sigma}}]$$

Transformation rules:

$$s_{\sigma}^{}$$
, = $s_{\sigma}^{}$

$$L_{\sigma}^{,} = L_{\sigma}^{\circ W_{i}}[x]$$

II.3: [Transfer the last committed value of a data item from the "materialized database" to the "audit trail"]

Conditions:

$$\exists T_i \exists x [T_i = lcw_C(x)]$$

Transformation rules:

$$s_{\sigma}$$
, = s_{σ}

$$SWI_{\sigma'} = SWI_{\sigma} \cup \{(x, M_{L_{\sigma}}(W_{i}[x]), T_{i})\}$$

$$L_{\sigma'} = L_{\sigma}$$

II.4: [Overwrite the value of x in the "materialized database" with an
 uncommitted value of x]

Conditions:

$$\begin{split} \exists \mathtt{T}_{\mathtt{i}} \exists \mathtt{x} [\mathtt{W}_{\mathtt{i}} [\mathtt{x}] \in \mathsf{OP}_{\mathtt{L}_{\sigma}} \wedge \mathtt{C}_{\mathtt{i}}, \mathtt{A}_{\mathtt{i}} \not\in \mathsf{OP}_{\mathtt{L}_{\sigma}} \wedge \\ [\mathtt{T}_{\mathtt{j}} = \mathtt{lcw}_{\sigma} (\mathtt{x}) \Rightarrow (\mathtt{W}_{\mathtt{j}} [\mathtt{x}] <_{\mathtt{L}_{\sigma}} \mathtt{W}_{\mathtt{i}} [\mathtt{x}] \wedge (\mathtt{x}, \mathtt{CS}_{\sigma} (\mathtt{x}), \mathtt{T}_{\mathtt{j}}) \in \mathtt{SWI}_{\sigma})]] \end{split}$$

Transformation rules:

$$S_{\sigma}, (y) = \begin{cases} S_{\sigma}(y) & \text{if } y \neq x \\ M_{L_{\sigma}}(W_{i}[x]) & \text{if } y = x \end{cases}$$

$$SWI_{\sigma}, = SWI_{\sigma}$$

$$L_{\sigma}, = L_{\sigma}$$

II.5: [Commit a transaction]

Conditions:

$$\exists \mathbf{T}_{\mathbf{i}} [C_{\mathbf{i}}, \mathbf{A}_{\mathbf{i}} \notin \mathsf{OP}_{\mathbf{L}_{\mathcal{O}}} \land \forall \mathbf{x} [W_{\mathbf{i}}[\mathbf{x}] \in \mathsf{OP}_{\mathbf{L}_{\mathcal{O}}} \land \mathsf{1cw}_{\mathcal{O}}(\mathbf{x}) = \mathbf{T}_{\mathbf{j}} \land \\ W_{\mathbf{j}}[\mathbf{x}] <_{\mathbf{L}_{\mathcal{O}}} W_{\mathbf{i}}[\mathbf{x}] \Rightarrow \mathbf{S}_{\mathcal{O}}(\mathbf{x}) = \mathbf{M}_{\mathbf{L}_{\mathcal{O}}}(W_{\mathbf{i}}[\mathbf{x}])]$$

Transformation rules:

$$S_{\sigma'} = S_{\sigma}$$

$$SWI_{\sigma'} = SWI_{\sigma}$$

$$L_{\sigma'} = L_{\sigma} \cdot C_{i}$$

II.6: [Abort a transaction]

Conditions:

$$\exists T_i [C_i, A_i \notin OP_{L_{\sigma}}]$$

Transformation rules:

$$s_{\sigma}$$
, = s_{σ}

$$L_{\sigma}^{\prime} = L_{\sigma}^{\circ A}_{i}$$

II.7: [Restore value of a data item, updated by an aborted transaction, in the "materialized database"]

Conditions:

$$\exists T_i \exists x [A_i, W_i[x] \in OP_{L_G} \land S_G(x) = M_{L_G}(W_i[x])]$$

Transformation rules:

$$S_{\sigma}, (y) = \begin{cases} S_{\sigma}(y) & \text{if } y \neq x \\ v & \text{if } y = x, \text{ where } (x, v, lcw_{\sigma}(x)) \in SWI_{\sigma} \end{cases}$$

$$SWI_{\sigma}, = SWI_{\sigma}$$

$$L_{\sigma} = L_{\sigma}$$

II.8: [Delete unnecessary information from the "audit trail"]

Conditions:

$$\exists T_i \exists x [T_i \neq lcw_G(x)]$$

Transformation rules:

$$S_{\sigma}$$
, = S_{σ}
 SWI_{σ} , = $SWI_{\sigma} - \{(x, v, T_{i}) | v \in HU\}$
 L_{σ} , = L_{σ} .

⁰3.5

THEOREM 3.6. Algorithm μ_{II} is correct.

<u>Proof.</u> Let $\sigma_0 \sigma_1 \dots \sigma_n$ be a μ -compatible history. We show, by induction on k, that σ_k is resilient for $0 \le k \le n$.

Basis: By Lemma 2.4, σ_0 is resilient.

Induction Step: Suppose σ_k is resilient for some $0 \le k \le n$. By μ_{II} -compatibility of $\sigma_0 \sigma_1 \dots \sigma_n$ we have that $\sigma_{k+1} \in \mu_p(\sigma_k)$. Consider each transition type.

Types II.1-3, and II.6: In all these cases we have $S_{\sigma_{k+1}} = S_{\sigma_k}$, and $SWI_{\sigma_{k+1}} \supseteq SWI_{\sigma_k}$. Also, since $L_{\sigma_{k+1}} \supseteq L_{\sigma_k}$ and exactly the same transactions are committed in the two logs (i.e., $Com(L_{\sigma_k}) = Com(L_{\sigma_k})$), it follows that $CS_{\sigma_{k+1}} = CS_{\sigma_k}$. Thus, if (R1) or (R2) hold for x in σ_k they also hold for x in σ_{k+1} . By induction hypothesis then, σ_{k+1} is resilient.

Type II.4: Let $x \in D$ be the data item such that $S_{\sigma_{k+1}}(x) \neq S_{\sigma_k}(x)$. Note that for all $y \in D - \{x\}$, if (R1) or (R2) hold for y in σ_k it also holds for y in σ_{k+1} . By inductive hypothesis then, it suffices to show that σ_{k+1} is resilient for x. By the conditions on II.4-type transitions we have that $(x,CS_{\sigma_k}(x),lcw_{\sigma_k}(x)) \in SWI_{\sigma_k}$. Since $L_{\sigma_{k+1}} = L_{\sigma_k}$ we have $lcw_{\sigma_k} = lcw_{\sigma_k}$ and $lcw_{\sigma_k} = lcw_{\sigma_k}$. Also, by the II.4-type transformation $lcw_{\sigma_k} = lcw_{\sigma_k}$ and $lcw_{\sigma_k} = lcw_{\sigma_k}$. Also, by the II.4-type transformation

rules we have $SWI_{\sigma_{k+1}} = SWI_{\sigma_k}$. Therefore, $(x,CS_{\sigma_{k+1}}(x),lcw_{\sigma_{k+1}}(x)) \in SWI_{\sigma_{k+1}}$. Thus (R2) holds for x in σ_{k+1} .

Type II.5: Let T_i be the transaction that committed in the transition from σ_k to σ_{k+1} (i.e. T_i is such that $L_{\sigma_{k+1}} = L_{\sigma_k} \circ C_i$). Consider any $x \in D$. There are two cases.

Case 2: $T_i \neq lcw_{0k+1}$ (x). It is then easy to see that lcw_{0k+1} (x) = lcw_{0k} (x) and CS_{0k+1} (x) = CS_{0k} (x). Since, by the type II.5 transformation rules, $S_{0k+1} = S_{0k}$ and $SWI_{0k+1} = SWI_{0k}$, (R1) or (R2) holds for x in S_{0k+1} if it holds for x in S_{0k+1} is resilient.

Type II.7: Let T_i , x satisfy the II.7 conditions. Since $A_i \in OP_L$, G_k clearly $T_i \neq lcw_{G_k}$ (x). Since S_{G_k} (x) = $M_{L_{G_k}}$ ($W_i[x]$) and $T_i \neq lcw_{G_k}$ (x), CS_{G_k} (x) = S_{G_k} (x) = S_{G_k} (x) and S_{G_k} (x). That is, (R1) is not satisfied for x in S_k . Because S_k is resilient, by inductive hypothesis, (R2) must be satisfied for x in S_k , i.e. (x, CS_{G_k} (x), lcw_{G_k} (x)) $\in SWI_{G_k}$. Since by the II.7 transformation rules S_{G_k+1} = S_{G_k} , we have that S_{G_k+1} = S_{G_k} and S_{G_k+1} = S_{G_k} .

Also, by the same transformation rules, $SWI_{\sigma_{k+1}} = SWI_{\sigma_k}$. Thus $(x,CS_{\sigma_{k+1}}(x), lcw_{\sigma_{k+1}}(x)) \in SWI_{\sigma_{k+1}}$, proving that (R2) holds for x in σ_{k+1} , i.e. σ_{k+1} is resilient for x. For all $y \in D - \{x\}$, it is easy to see that (R1) or (R2) holds for y in σ_{k+1} if it does in σ_k . By inductive hypothesis then σ_{k+1} is resilient for $y \in D - \{x\}$, as well.

Type II.8: Let T_i , x satisfy the conditions of II.8 transition type. Clearly, for any $y \in D - \{x\}$, if (R1) or (R2) holds for y in σ_k it also holds in σ_{k+1} , and by inductive hypothesis then, σ_{k+1} is resilient for such y. Also if (R1) is satisfied for x in σ_k it will be satisfied in σ_{k+1} . The only case of concern then, is if (R2) is satisfied for x in σ_k . Then $(x,CS_{\sigma_k}(x), lcw_{\sigma_k}(x)) \in SWI_{\sigma_k}$. By the II.8 conditions, $T_i \neq lcw_{\sigma_k}(x), \text{ hence } (x,CS_{\sigma_k}(x), lcw_{\sigma_k}(x)) \in SWI_{\sigma_{k+1}}$. Also, because $L_{\sigma_{k+1}} = L_{\sigma_k}, lcw_{\sigma_{k+1}} = lcw_{\sigma_k} \text{ and } CS_{\sigma_k} = CS_{\sigma_k}. \text{ Therefore,}$ $(x,CS_{\sigma_{k+1}}(x), lcw_{\sigma_{k+1}}(x)) \in SWI_{\sigma_k} = stablishing that (R2) holds for <math>x$ in σ_{k+1} . So σ_{k+1} is resilient for x, too.

This concludes the Induction Step and thereby the proof. $_{3.6}$

THEOREM 3.7. Algorithm μ_{II} never requires redo but may require undo.

 that the transition from σ_{k-1} to σ_k can only be of type II.4. Let $T_j = lcw_{\sigma_k}(x)$ (3.7.2). By the conditions of II.4 transition type we have that $W_j[x] <_{L_{\sigma_{k-1}}} W_i[x]$ (3.7.3). Also, by the same conditions we have that $C_i \not\in OP_{L_{\sigma_{k-1}}}$ and hence $C_i \not\in OP_{L_{\sigma_k}}$ (recall that the transition from σ_{k-1} to σ_k is of type II.4, so T_i can't commit during that transition). Because, by assumption, $C_i \in OP_{L_{\sigma_k}}$, we must have that for some $p, k , <math>L_{\sigma_k} = L_{\sigma_{p-1}} \circ C_i$ (3.7.4).

Claim: For any $q \neq p$, $k < q \leq n$, $lcw_{q}(x) = lcw_{q-1}(x)$.

Proof (of the Claim): By definition of lcw, lcw $_{Q}(x) \neq lcw _{Q-1}(x)$, only if $L_{Q} = L_{Q} \circ C_{S}$ and $W_{S}[x] \in OP_{L_{Q}}$, i.e. only if in the transition $Q_{Q}(x) = Q_{Q}(x)$ from $Q_{Q}(x) = Q_{Q}(x)$ a transaction $Q_{Q}(x) = Q_{Q}(x)$ and $Q_{Q}(x) = Q_{Q}(x)$ and $Q_{Q}(x) = Q_{Q}(x)$ and $Q_{Q}(x) = Q_{Q}(x)$ and the type $Q_{Q}(x) = Q_{Q}(x)$ and the case because, by (3.7.1) $Q_{Q}(x) = Q_{Q}(x)$ for all $Q_{Q}(x) = Q_{Q}(x)$ and since $Q_{Q}(x) = Q_{Q}(x)$ and the case because have that $Q_{Q}(x) = Q_{Q}(x)$ and since $Q_{Q}(x) = Q_{Q}(x)$ and $Q_{Q}(x) = Q_{Q}(x)$ and therefore $Q_{Q}(x) = Q_{Q}(x)$. Therefore, we must have that $Q_{Q}(x) = Q_{Q}(x)$ and since $Q_{Q}(x) = Q_{Q}(x)$ and therefore $Q_{Q}(x) = Q_{Q}(x)$.

Now, by using (3.7.2), the Claim, and induction on q for $k \le q < p$, we get that $lcw_{\sigma_{p-1}}(x) = T_j$. By (3.7.4) $C_i \in OP_L$, and by (3.7.3) (and the fact that $L_{\sigma_p} \supseteq L_{\sigma_{k-1}}$) we have that $lcw_{\sigma_p}(x) = T_i$. By using this,

the Claim and induction on q for $p < q \le n$, we get that $lcw_{0}(x) = T_{1}$, as wanted.

To show that μ_{II} may require undo, consider the history $\sigma_0\sigma_1\sigma_2\sigma_3$ defined as follows:

It can be easily checked that this is a μ_{II} -compatible history (σ_0 goes to σ_1 by means of a type II.2 transition, σ_1 to σ_2 by means of a type II.3 transition, and σ_2 to σ_3 by means of a type II.4 transition), and that $S_{\sigma_3}(\mathbf{x}) = \hat{g}_{1\mathbf{x}}(\cdot) = M_{L_{\sigma_3}}(W_1[\mathbf{x}])$, while $C_1 \not\in OP_{L_{\sigma_3}}$ (no type 5 transitions). $\sigma_0 \sigma_1 \sigma_2 \sigma_3$ shows that μ_{II} may require undo.

THEOREM 3.8. Algorithm μ_{II} is well-formed.

<u>Proof.</u> That μ_{II} satisfies WFA1 follows directly from transition types II.1,2 and 6. We proceed to show that μ_{II} satisfies WFA2. Consider any μ_{II} -compatible history $\alpha = \sigma_0 \sigma_1 \dots \sigma_n$. Let T_i be any transaction such that A_i , $C_i \not\in \text{OP}_{L_o}$.

Case 1: If, moreover, for all $W_i[x] \in OP_{L_{\sigma_n}}$ such that $W_j[x] <_{L_{\sigma_n}} W_i[x]$, where $\bar{T}_j = lcw_{\sigma_n}(x)$, it is the case that $S_{\sigma_n}(x) = M_{L_{\sigma_n}}(W_i[x])$, the II.5

conditions are satisfied and we may define $\tau_1 \in \mu_{II}(\sigma_n)$ by a type II.5 transition. Taking $\beta = \tau_1$, we have that $\alpha\beta$ is μ_{II} -compatible and $L_{\tau_1} = L_{\sigma} \circ C_i$, proving that WFA2 holds, in this case.

Case 2: Now suppose that there exists some $x \in D$ such that $W_j[x] <_{L_{\sigma_n}} W_i[x]$, where $T_j = lcw_{\sigma_n}(x)$, yet $S_{\sigma_n}(x) \neq M_{L_{\sigma_n}}(W_i[x])$ (3.8.1). Let x_1, x_2, \ldots, x_m be all such x's. Define history $T_1 T_2 \ldots T_{2m+1}$ as follows:

Let $T_0 \equiv \sigma_n$, and denote $T_j(x_k) = lcw_{\sigma_n}(x_k)$, for $1 \le k \le m$.

Let

$$S_{\tau_{k}} = S_{\tau_{k-1}}$$

$$SWI_{\tau_{k}} = SWI_{\tau_{k-1}} \cup \{(x_{k}, M_{L_{\sigma_{k-1}}}(W_{j}(x_{k}), x_{j}), T_{j}(x_{k})\}\}$$

$$L_{\tau_{k}} = L_{\tau_{k-1}}, \quad \text{for } 1 \le k \le m.$$

It is easy to see that the transition from τ_{k-1} to τ_k is of type II.3 and at each step the conditions are enabled. So the history $\sigma_0\sigma_1\dots\sigma_n\tau_1\tau_2\dots\tau_m$ is μ_{II} -compatible. Also, it can be easily verified that $(\mathbf{x}_k, CS_{\tau_m}(\mathbf{x}_k), T_{\tau_m}(\mathbf{x}_k))$ and that $T_{\mathbf{j}}(\mathbf{x}_k) = 1cw_{\tau_m}(\mathbf{x}_k)$, for $1 \le k \le m$. By choice of the \mathbf{x}_k 's, and the fact that $\mathbf{L}_{\tau_m} = \mathbf{L}_{\sigma_m}$ we get that $\mathbf{W}_{\mathbf{j}}(\mathbf{x}_k) = \mathbf{L}_{\tau_m} \mathbf{W}_{\mathbf{j}}(\mathbf{x}_k)$, $\mathbf{L}_{\tau_m} \mathbf{W}_{\mathbf{j}}(\mathbf{x}_k)$, $\mathbf{L}_{\tau_m} \mathbf{W}_{\mathbf{j}}(\mathbf{x}_k)$, $\mathbf{L}_{\tau_m} \mathbf{W}_{\mathbf{j}}(\mathbf{x}_k)$, the conditions for II.4 transitions are enabled for $T_{\mathbf{j}}$ and the \mathbf{x}_k 's. So, we let

$$S_{\tau_{m+k}}(y) = \begin{cases} S_{\tau_{m+k-1}}(y) & \text{if } y \neq x_k \\ M_{L_{\tau_{m+k-1}}}(W_i[x_k]) & \text{if } y = x_k \end{cases}$$

$$SWI_{\tau_{m+k}} = SWI_{\tau_{m+k-1}}$$

$$L_{\tau} = L_{\tau}$$
, for $1 \le k \le m$.

It can be seen that $\sigma_0\sigma_1\dots\tau_1\tau_2\dots\tau_{2m}$ is μ_{II} -compatible, that $1\text{cw}_{\tau_{2m}}(\mathbf{x}_k)=\mathrm{T}_{\mathbf{j}}(\mathbf{x}_k)$, $\mathbb{W}_{\mathbf{j}}(\mathbf{x}_k)$ $[\mathbf{x}_k]$ $[\mathbf{x}_k]$ and $S_{\sigma_{\tau_{2m}}}(\mathbf{x}_k)=\mathrm{M}_{L_{\tau_{2m}}}(\mathbb{W}_{\mathbf{j}}[\mathbf{x}_k])$ for $1\leqslant k\leqslant m$. Since the \mathbf{x}_k 's were the only data items for which the II.5 conditions were violated on σ_n and are no longer violated on τ_{2m} , we can define a type II.5 transition from τ_{2m} to τ_{2m+1} so that

$$S_{\tau} = S_{\tau},$$

$$SWI_{\tau} = SWI_{\tau}, \text{ and}$$

$$L_{\tau} = L_{\tau} \circ C_{i}.$$

Let $\beta=\tau_1\tau_2...\tau_{2m+1}$. We have proved that $\alpha\beta$ is μ_{II} -compatible and since L=L we also have that L=L = L oC, finally proving that WFA2 holds in this case, too.

3.4 Algorithm III (no undo but redo)

ALGORITHM 3.9. $\sigma' \in \mu_{\text{TTT}}(\sigma)$ iff one of the following is the case:

III.1: [Submit a read operation]

Conditions:

$$\exists T_i \exists x [C_i, A_i, R_i[x] \notin OP_{L_C}]$$

Transformation rules:

$$S_{\sigma}^{\dagger} = S_{\sigma}^{\dagger}$$

$$SWI_{\sigma}^{\dagger} = SWI_{\sigma}^{\dagger}$$

$$L_{\sigma}^{\dagger} = L_{\sigma}^{\dagger} R_{i}^{\dagger} [x]$$

III.2: [Submit a write operation]

Conditions:

$$\exists T_i \exists x [C_i, A_i, W_i[x] \notin OP_{L_C}]$$

Transformation rules:

$$S_{\sigma}^{i} = S_{\sigma}^{i}$$

$$SWI_{\sigma}^{i} = SWI_{\sigma}^{i}$$

$$L_{\sigma}^{i} = L_{\sigma}^{o}W_{i}^{i}[x]$$

III.3: [Record an uncommitted update to "audit trail"--create "intention
list" |]

Conditions:

$$\exists T_i \exists x [C_i, A_i \notin OP_{L_{\sigma}} \land W_i [x] \in OP_{L_{\sigma}}]$$

Transformation rules:

$$S_{\sigma}$$
, = S_{σ}
 SWI_{σ} , = $SWI_{\sigma} \cup \{(x, M_{L_{\sigma}}(W_{i}[x]), T_{i})\}$
 L_{σ} , = L_{σ}

$$\exists \mathtt{T_{i}} \left[\mathtt{C_{i}}, \mathtt{A_{i}} \notin \mathtt{OP}_{\mathtt{L_{o}}} \land \forall \mathtt{W_{i}} \left[\mathtt{x}\right] \in \mathtt{OP}_{\mathtt{L_{o}}} \left[\left(\mathtt{x}, \mathtt{M}_{\mathtt{L_{o}}} (\mathtt{W_{i}} \left[\mathtt{x}\right]\right), \mathtt{T_{i}}\right) \in \mathtt{SWI_{o}}\right]\right]$$

In the earliest description of a class III algorithm (that we know of) [LS76] the data structure used in the audit trail is called the "intention list". The term derives from the fact that a transaction creates a list of the updates it "intends" to perform, writes this list in stable storage (the "audit trail") and then commits later the intentions list is carried out and the updates recorded in the materialized database.

Transformation rules:

$$S_{\sigma'} = S_{\sigma}$$

$$SWI_{\sigma'} = SWI_{\sigma}$$

$$L_{\sigma'} = L_{\sigma} \cdot C_{i}$$

III.5: [Move updates of a committed transaction from the "audit trail"
 to the "materialized database", one at a time--i.e. start carrying
 out "intentions list"]

Conditions:

$$\exists \mathtt{T_i} \exists \mathtt{x} [\mathtt{T_i} = \mathtt{lcw}_{\sigma}(\mathtt{x}) \land (\mathtt{x}, \mathtt{M}_{\mathtt{L}_{\sigma}}(\mathtt{W_i}[\mathtt{x}]), \mathtt{T_i}) \in \mathtt{SWI}_{\sigma}]$$

Transformation rules:

$$S_{\sigma}, (y) = \begin{cases} S_{\sigma}(y) & \text{if } y \neq x \\ M_{L_{\sigma}}(W_{\underline{i}}[x]) & \text{if } y = x \end{cases}$$

$$swi_{\sigma}$$
, = swi_{σ}

$$L_{\sigma}$$
, = L_{σ}

 $\underline{\text{III.6}}$: [Delete "intentions list" of transaction T_i once it has been carried out or T_i aborted]

Conditions:

$$\exists \mathtt{T}_{\mathtt{i}} \, [\, (\mathtt{C}_{\mathtt{i}} \, \in \mathtt{OP}_{\mathtt{L}_{\mathtt{O}}} \, \land \, \forall \mathtt{W}_{\mathtt{i}} \, [\mathtt{x}] \, \in \mathtt{OP}_{\mathtt{L}_{\mathtt{O}}} [\mathtt{S}_{\mathtt{O}}(\mathtt{x}) = \mathtt{M}_{\mathtt{L}_{\mathtt{O}}} (\mathtt{W}_{\mathtt{i}} \, [\mathtt{x}]) \,]) \, \vee \, \mathtt{A}_{\mathtt{i}} \, \in \mathtt{OP}_{\mathtt{L}_{\mathtt{O}}}]$$

Transformation rules:

$$S_{\sigma}^{\prime} = S_{\sigma}^{\prime}$$

$$SWI_{\sigma}^{\prime} = SWI_{\sigma}^{\prime} - \{(x, M_{L_{\sigma}^{\prime}}(W_{i}[x]), T_{i}) | x \in D\}$$

$$L_{\sigma}^{\prime} = L_{\sigma}^{\prime}$$

III.7: [Abort a transaction]

Conditions:

$$\exists T_i[c_i, A_i \notin OP_{L_G}]$$

Transformation rules:

$$s_{\sigma} = s_{\sigma}$$

SWI_o, = SWI_o

$$L_{\sigma}^{\dagger} = L_{\sigma}^{\bullet} A_{i}$$
.

⁰3.9

THEOREM 3.10. Algorithm μ_{III} is correct.

<u>Proof.</u> Let $\sigma_0 \sigma_1 \dots \sigma_n$ be a μ_{III} -compatible history. We'll show, by induction on k, that σ_i is resilient for $0 \le k \le n$.

Basis: σ_0 is resilient, by Lemma 2.4.

Induction Step: Suppose σ_k is resilient for some $0 \le k \le n$. By μ_{III} -compatibility of $\sigma_0 \sigma_1 \dots \sigma_n$ we have that $\sigma_{k+1} \in \mu_{\text{III}}(\sigma_k)$. Consider the type of transition from σ_k to σ_{k+1} .

Types III.1-3 and 7: In all these cases we have that lcw_{σ} , = lcw_{σ} , CS_{σ} , = CS_{σ} , S_{σ} , = S_{σ} and SWI_{σ} , $\supseteq SWI_{\sigma}$. Thus, if (R1) or (R2) holds for some $x \in D$ in σ_k it also holds for x in σ_{k+1} . By induction hypothesis then σ_{k+1} is resilient in this case.

Type III.4: Let T_i be the transaction that commits in the transition from σ_k to σ_{k+1} ; i.e. $L_{\sigma_{k+1}} = L_{\sigma_k} \circ C_i$. Consider any $x \in D$.

(for (R1) note that $S_{\sigma_{k+1}} = S_{\sigma_{k}}$ and for (R2) that $SWI_{\sigma_{k+1}} = SWI_{\sigma_{k}}$, by the III.4 transformation rules). By induction hypothesis then, σ_{k+1} is resilient for x, in this case

Type III.5: Let T_i , x be such that $S_{\sigma_{k+1}}(x) = M_{L_{\sigma_k}}(W_i[x]) \neq S_{\sigma_k}(x)$. Consider any $y \in D$.

Case 1: $y \neq x$. Then $S_{\sigma_{k+1}}(y) = S_{\sigma_{k}}(y)$ and $SWI_{\sigma_{k+1}} = SWI_{\sigma_{k}}$, by the III.5 transformation rules. Since also $1cw_{\sigma_{k+1}} = 1cw_{\sigma_{k}}$ and $CS_{\sigma_{k+1}} = CS_{\sigma_{k}}$, it follows that if (R1) or (R2) holds for y on σ_{k} it also holds for y in σ_{k+1} . By induction hypothesis then, σ_{k+1} is resilient for y, in this case.

Case 2: y = x. Then, by the III.5 conditions, $T_i = lcw_{O_k}(x)$ and $(x,M_{L_O}(w_i[x]),T_i) \in SWI_{O_k}$. Since $lcw_{O_{k+1}} = lcw_{O_k}$, $CS_{O_{k+1}}(x) = M_{L_{O_{k+1}}(x)}(w_i[x])$ and $SWI_{O_{k+1}} = SWI_{O_k}$, we have that $(x,CS_{O_{k+1}}(x),L_{O_{k+1}(x)}(x))$ establishing that (R^2) holds for x in O_{k+1} . Thus O_{k+1} is resilient for x = y in this case, also.

Type III.6: Let T_i be such that for all $W_i[x] \in OP_{L_{O_k}}$, $(x,M_{L_{O_k}}(W_i[x]),T_i) \in SWI_{O_k} - SWI_{O_{k+1}}$. It is clear that if (R1) holds for any $x \in D$ in O_k it also holds for x in O_{k+1} , since $CS_{O_{k+1}} = CS_{O_k}$ and $S_{O_{k+1}} = S_{O_k}$. So, suppose that (R2) holds for x in O_k . That is, $(x,CS_{O_k}(x), 1cw_{O_k}(x)) \in SWI_{O_k}$. If $T_i \neq 1cw_{O_k}(x)$ then $(x,CS_{O_k}(x), 1cw_{O_k}(x)) \in SWI_{O_k}$ and since $CS_{O_{k+1}} = CS_{O_k}$ and $1cw_{O_k+1} = 1cw_{O_k}$, (R2) holds for x in O_{k+1} , also. If $T_i = 1cw_{O_k}(x)$, $A_i \notin OP_{L_{O_k}}$ and by the III.6 conditions, we must have that $S_{O_k}(x) = M_{L_{O_k}}(w_i[x]) = CS_{O_k}(x)$. Because $S_{O_{k+1}} = S_{O_k}$ and $CS_{O_{k+1}} = CS_{O_k}$, we get $S_{O_{k+1}}(x) = CS_{O_k}(x)$ and thus (R1) holds for x in O_{k+1} . Therefore, in either case $(T_i = 1cw_{O_k}(x))$ or $T_i \neq 1cw_{O_k}(x)$) O_{k+1} is resilient for x.

This completes the induction step and thereby the proof. 3.10

THEOREM 3.11. Algorithm μ_{III} never requires undo but may require redo.

Proof. Let $\sigma_0\sigma_1...\sigma_n$ be any μ_{III} -compatible history. Consider any $\mathbf{x} \in \mathbb{D}$ and let $S_{\sigma_n}(\mathbf{x}) = M_{\mathbf{L}_{\sigma_n}}(W_{\mathbf{i}}[\mathbf{x}])$. To prove that μ_{III} never requires undo, it suffices to show that $C_{\mathbf{i}} \in \mathsf{OP}_{\mathbf{L}_{\sigma_n}}$. Indeed, let $\mathbf{j} \leq n$ be such that $S_{\sigma_{\mathbf{j}-1}}(\mathbf{x}) \neq M_{\mathbf{L}_{\sigma_n}}(W_{\mathbf{i}}[\mathbf{x}])$ and for $\mathbf{j} \leq \mathbf{k} \leq n$, $S_{\sigma_n}(\mathbf{x}) = M_{\mathbf{L}_{\sigma_n}}(W_{\mathbf{i}}[\mathbf{x}])$. If no such \mathbf{j} exists, $S_{\sigma_n}(\mathbf{x}) = M_{\mathbf{L}_{\sigma_n}}(W_{\mathbf{0}}[\mathbf{x}])$ and $C_{\mathbf{0}} \in \mathsf{OP}_{\mathbf{L}_{\sigma_n}}$, and we are done. By inspection of the transition rules of μ_{III} , we have that the transition from $\sigma_{\mathbf{j}-1}$ to $\sigma_{\mathbf{j}}$ can only be of type III.5 (it's the only type in which the S-component of the system state changes). By the III.5 conditions $T_{\mathbf{i}} = \mathbf{lcw}_{\sigma_{\mathbf{j}-1}}(\mathbf{x})$ and thus $C_{\mathbf{i}} \in \mathsf{OP}_{\mathbf{L}_{\sigma_n}}$. Surely then $C_{\mathbf{i}} \in \mathsf{OP}_{\mathbf{L}_{\sigma_n}}(\mathbf{x}) = \mathbf{lcw}_{\sigma_{\mathbf{j}-1}}(\mathbf{x})$, as required.

To see that $~\mu_{III}~$ may require redo, consider the history ${\sigma_0}{\sigma_1}{\sigma_2}{\sigma_3}~$ defined as follows:

 $(\sigma_0$ is the initial system state)

$$S_{\sigma_{1}} = S_{\sigma_{0}}, \quad SWI_{\sigma_{1}} = SWI_{\sigma_{0}}, \quad L_{\sigma_{1}} = L_{\sigma_{0}} \circ W_{1}[x]$$

$$S_{\sigma_{2}} = S_{\sigma_{1}}, \quad SWI_{\sigma_{2}} = SWI_{\sigma_{1}} \cup \{(x, M_{L_{\sigma_{1}}}(W_{1}[x]), T_{1})\}, \quad L_{\sigma_{2}} = L_{\sigma_{1}}$$

$$S_{\sigma_{3}} = S_{\sigma_{2}}, \quad SWI_{\sigma_{3}} = SWI_{\sigma_{2}}, \quad L_{\sigma_{3}} = L_{\sigma_{2}} \circ C_{1}$$

It can be readily verified that this is a μ_{III} -compatible history (σ_0 goes to σ_1 by a III.2 transition, σ_1 to σ_2 by a III.3 transition and σ_2 to σ_3 by a III.4 transition), that $S_{\sigma_3}(x) = \hat{g}_{10}(\cdot) = M_{L_{\sigma_3}}(W_0[x])$, $C_0 \in \text{OP}_{L_{\sigma_3}}$, yet $\text{lcw}_{\sigma_3}(x) = T_1 \neq T_0$. $\sigma_0 \sigma_1 \sigma_2 \sigma_3$ shows that μ_{III} may require redo.

THEOREM 3.12. Algorithm μ_{III} is well-formed.

<u>Proof.</u> By transition types III.1,2 and 7 it is immediate that μ_{III} satisfies WFA1. We show that it also satisfies WFA2. Let $\alpha = \sigma_0 \sigma_1 \dots \sigma_n$ be any μ_{III} -compatible history. Consider any T_i such that C_i , $A_i \not\in \text{OP}_{L_{\sigma_n}}$

Case 1: If for all $W_i[x] \in OP_{L_{\sigma}}$, $(x,M_{L_{\sigma}}(W_i[x]),T_i) \in SWI_{\sigma}$, the III.4 conditions are satisfied and we may define $\tau_1 \in \mu_{III}(\sigma_n)$ such that $L_{\tau_1} = L_{\sigma} \circ C_i$. Since $\sigma_0 \sigma_1 \dots \sigma_n \tau_1$ is μ_{III} -compatible we have that μ_{III} satisfies WFA2, in this case.

Case 2: Suppose for some $W_i[x] \in OP_{L_{\overline{O}_n}}$, $(x, M_{L_{\overline{O}_n}}(W_i[x]), T_i) \notin SWI_{\overline{O}_n}$ (3.12.1). Let x_1, \dots, x_m be all such $x \in D$. Since C_i , $A_i \notin OP_{L_{\overline{O}_n}}$ and

 $W_i[x] \in OP_L$, the III.3 conditions are enabled for T_i and any x_k $1 \le k \le m$, so we may define T_k , $0 \le k \le m$ as follows:

$$\begin{array}{l} \tau_0 \equiv \sigma_n, \quad \text{and} \\ \\ S_{\tau_{k+1}} = S_{\tau_k} \\ \\ SWI_{\tau_{k+1}} = SWI_{\tau_k} \cup \{(x_k, M_{L_{\sigma_k}}(W_i[x_k]), \tau_i)\}, \quad \text{and} \\ \\ L_{\tau_{k+1}} = L_{\tau_k}, \quad \text{for } 0 \leq k \leq m \end{array}.$$

Since x_1, \dots, x_m were all the data items $x \in D$ such that (3.12.1) is true and clearly, C_i , $A_i \notin OP_{L_{T_m}}$ and $(x_k, M_{L_{T_m}}(W_i[x_k]), T_i) \in SWI_{T_m}$, the III.4 conditions are satisfied in T_m and we may thus define $T_{m+1} \in \mu_{LS}(T_m)$ by a III.4 transition. If $\beta = T_1 T_2 \dots T_{m+1}$, we have that $\alpha\beta$ is μ_{III} -compatible and $L_{T_{m+1}} = L_{O_n} \circ C_i$. Thus μ_{III} satisfies WFA2 in this case also.

3.5 Algorithm IV (neither undo nor redo)

ALGORITHM 3.13. $\sigma' \in \mu_{TV}(\sigma)$ if one of the following is the case:

IV.1: [Submit a read operation]

Conditions:

$$\exists T_i \exists x [C_i, A_i, R_i[x] \notin OP_{L_G}]$$

Transformation rules:

$$S_{\sigma}^{\dagger} = S_{\sigma}^{\dagger}$$

$$SWI_{\sigma}^{\dagger} = SWI_{\sigma}^{\dagger}$$

$$L_{\sigma}^{\dagger} = L_{\sigma}^{\dagger} R_{i}^{\dagger} [x]$$

IV.2: [Submit a write operation]

Conditions:

$$\exists T_i \exists x [C_i, A_i, W_i [x] \notin OP_{L_G}]$$

Transformation rules:

$$S_{\sigma}' = S_{\sigma}$$

$$SWI_{\sigma}' = SWI_{\sigma}$$

$$L_{\sigma}' = L_{\sigma} \circ W_{i}[x]$$

IV.3: [Record a submitted update on the "audit trail"]

Conditions:

$$\exists T_i \exists x [C_i, A_i \notin OP_{L_{\sigma}} \land W_i [x] \in OP_{L_{\sigma}}]$$

Transformation rules:

$$S_{\sigma}$$
, = S_{σ}
 SWI_{σ} , = $SWI_{\sigma} \cup \{(x, M_{L_{\sigma}}(W_{i}[x]), T_{i})\}$
 L_{σ} , = L_{σ}

$$\exists \mathtt{T_{i}} [\mathtt{C_{i}}, \mathtt{A_{i}} \not\in \mathtt{OP}_{\mathtt{L_{o}}} \land \forall \mathtt{x} \in \mathtt{D}[\mathtt{W_{i}}[\mathtt{x}] \in \mathtt{OP}_{\mathtt{L_{o}}} \rightarrow (\mathtt{x}, \mathtt{M}_{\mathtt{L_{o}}}(\mathtt{W_{i}}[\mathtt{x}]), \mathtt{T_{i}}) \in \mathtt{SWI}_{o}]]$$

Transformation rules:

$$= S_{\sigma}^{(x)} = \begin{cases} M_{L_{\sigma}}^{(W_{i}[x])}, & \text{if } W_{i}[x] \in OP_{L_{\sigma}} \land lcw_{\sigma}(x) = T_{j} \Rightarrow W_{j}[x] <_{L_{\sigma}} W_{i}[x] \\ \\ S_{\sigma}^{(x)} & \text{otherwise} \end{cases}$$

$$L_{\sigma}$$
, = $L_{\sigma} \circ C_{i}$

IV.5: [Delete unnecessary information from the "audit trail"]

Conditions:

$$\exists \mathtt{T_{i}} [\mathtt{C_{i}} \in \mathtt{OP}_{\mathtt{L}_{\sigma}} \lor \mathtt{A_{i}} \in \mathtt{OP}_{\mathtt{L}_{\sigma}}]$$

Transition rules:

$$s_{\sigma}$$
, = s_{σ}

$$SWI_{\sigma}$$
 = SWI_{σ} - { $(x,v,T_i) | x \in D, v \in HU$ }

$$L_{\sigma} = L_{\sigma}$$

IV.6: [Abort transaction T;]

Conditions:

Transformation rules:

$$s_{\sigma}$$
, s_{σ}

$$L_{\sigma}$$
, = $L_{\sigma} \cdot A_{i}$

²3.13

THEOREM 3.14. Algorithm μ_{TV} is correct.

<u>Proof.</u> Let $\sigma_0 \sigma_1 \dots \sigma_n$ be a μ_{IV} -compatible history. We'll show, by induction, that σ_1 is resilient for $0 \le i \le n$. In fact, we'll show something even stronger, namely that each σ_i satisfies condition (R1) for all $x \in D$.

Basis: By the proof of Lemma 2.4, σ_0 satisfies (R1).

Induction Step: Let $0 \le k \le n$ and suppose σ_k satisfies (R1). Since $\sigma_0 \sigma_1 \dots \sigma_n$ is μ_{IV} -compatible we know $\sigma_{k+1} \in \mu_{IV}(\sigma_k)$. We'll show that σ_{k+1} satisfies (R1) by considering each type of transition defined by μ_{IV} .

Types IV.1-3, IV.5-6: In all these cases, $L_{\sigma_{k+1}} \supseteq L_{\sigma_{k}}$, and the same transactions are committed in both logs $(Com(L_{\sigma_{k+1}}) = Com(L_{\sigma_{k}}))$. Therefore $CS_{\sigma_{k+1}} = CS_{\sigma_{k}}$. Also, in all these cases $S_{\sigma_{k+1}} = S_{\sigma_{k}}$. By inductive hypothesis, $S_{\sigma_{k}} = CS_{\sigma_{k}}$. Hence, $S_{\sigma_{k+1}} = CS_{\sigma_{k+1}}$ as wanted.

Type IV.4: Let T_i be the transaction that was committed in the transition from σ_k to σ_{k+1} ; i.e. T_i is such that $L_{\sigma_{k+1}} = L_{\sigma_k} \circ C_i$. Consider any $x \in D$. We want to show that $S_{\sigma_{k+1}} = CS_{\sigma_k} = CS_{$

This concludes the induction step and thereby the proof. $\Box_{3,14}$

THEOREM 3.15. Algorithm μ_{TV} never requires redo or undo.

<u>Proof.</u> In 3.14 we actually proved that for all μ_L -compatible histories $\sigma_0 \sigma_1 \dots \sigma_n$, for all $0 \le k \le n$ all $x \in D$, $S_{\sigma_k}(x) = CS_{\sigma_k}(x)$. Definitions 2.6 and 2.7 then, immediately yield the theorem.

Proof. WFAl is immediate from transition types IV.1,2 and 6.

THEOREM 3.16. Algorithm μ_{IV} is well-formed.

So we only need to show that μ_{IV} satisfies WFA2. Let $\alpha = \sigma_0 \sigma_1 \dots \sigma_n$ be any μ_{IV} -compatible history and consider any T_i such that C_i , $A_i \notin \text{OP}_{L_{\sigma_n}}$.

Case 1: If it is also the case that for all $W_i [x] \in \text{OP}_{L_{\sigma_n}}$, $(x, M_{L_{\sigma_n}}(W_i [x]), T_i) \in \text{SWI}_{\sigma_n}$ the conditions of type IV.4 transitions are enabled and so we may define $T_1 \in \mu_{\text{IV}}(\sigma_n)$ such that T_1 is obtained from σ_n by the transformation rules of type IV.4 transitions. By defining $\beta = T_1$, we get that $\alpha\beta$ is μ_{IV} -compatible and $L_{T_1} = L_{\sigma_n} \circ C_i$, with C_i , $A_i \notin \text{OP}_{L_{\sigma_n}}$ verifying that μ_{IV} satisfies WFA2, in this case.

Case 2: Suppose, on the other hand, that for some $x \in D$, $W_i[x] \in OP_{L_{O_n}}$ but $(x,M_{L_{O_n}}(W_i[x]),T_i) \notin SWI_{O_n}$. Let $x_1,x_2,\ldots,x_m \in D$ be all such x's. Then define $\beta = \tau_1\tau_2\ldots\tau_{m+1}$ where

$$SWI_{\tau_{1}} = SWI_{\sigma_{n}} \cup \{(x_{1}, M_{L_{\sigma_{n}}}(W_{i}[x_{1}]), T_{i})\}$$

$$SWI_{\tau_{k}} = SWI_{\sigma_{k-1}} \cup \{(x_{k}, M_{L_{\sigma_{k-1}}}(W_{i}[x_{k}]), T_{i})\} \qquad \text{for } 2 \leq k \leq m,$$

and

$$L_{\tau_{m+1}} = L_{\tau_m} \circ C_i$$

Clearly, $\tau_1 \in \mu_{IV}(\sigma_n)$ and $\tau_k \in \mu_{IV}(\tau_{k-1})$ for $2 \le k \le m$ by means of type IV.3 transitions. Note that $W_1[x_1] \in L_{\sigma_n}$, C_1 , $A_1 \in OP_{L_{\sigma_n}}$ and $W_1[x_k] \in OP_{L_{\tau_{k-1}}}$ and C_1 , C_1 , C_2 , C_3 , C_4 , C_5 , C_5 , C_6 , C_7 , C_8 , C_8 , so that such transitions are enabled. Also, C_1 , C_1 , C_2 , C_3 , C_4 , C_5 , C_6 , C_7 , C_8 , and C_8 , C_8 , C_8 , C_8 , C_8 , C_8 , C_8 , and C_8 , C_8 , C_8 , C_8 , and C_8 , and C_8 , C_8 , and C_8 , are also as a constant and C_8 , and C_8 , and C_8 , are also

4. DISTRIBUTED DATABASE SYSTEMS

4.1 Introduction

In this section we extend our model to describe certain aspects of reliable processing of operations in the context of a distributed database system. Speaking in broad terms, a distributed database system consists of a number of independent processors, called sites, each of which stores one or more data items. Transactions wishing to access the data items submit operations to the appropriate sites (usually through a scheduler). These operations are processed locally (independently of what happens at other sites). Eventually each transaction must be terminated by either becoming committed, or aborting. This decision is reached by consensus of the sites at which the transaction submitted operations: if all these sites agree to commit it, the transaction is globally committed; if even one site dissents, the transaction must be globally aborted.

This consensus voting scheme is called atomic commitment and an algorithm that implements it is called an atomic commitment algorithm (protocol). Several such algorithms have been proposed, for example two-phase commit ([LS76], [G78]), three-phase commit ([S82]), the SDD-1 (four-phase) commit algorithm ([HS80]) among others.

Atomic commitment is a coordination problem and, therefore, inherently involves communication among the participating (voting) sites. Because sites are independent processors (and may, therefore, fail independently) and because the medium of inter-site communication is also subject to a variety of failures, there are new kinds of fault tolerance that we might like a distributed database system to provide. Indeed, the various atomic

commitment algorithms mentioned above have different properties of fault tolerance with respect to these new types of failures. Some interesting work on this question can be found in [S82].

Description of atomic commitment algorithms and proofs of their fault tolerance properties is beyond the scope of the present paper. We therefore deliberately extend our model in a way that factors out these questions. (Other extensions, with these questions in mind, are possible and will be sketched in the next section.)

Rather, we are interested in understanding how to build reliable distributed algorithms for processing database operations out of reliable centralized such algorithms, as described in Sections 2 and 3, under the assumptions that the communication medium does not fail and that failures of sites are detected by other sites. As we shall see, it is possible to build "heterogeneous" distributed algorithms—in the sense that different sites may use different centralized algorithms.

Our primary goal is the description of the conditions under which a site should vote to commit or abort a transaction in an atomic commitment protocol. This will be done by giving a predicate $VOTE(\sigma^i,T_k)$ where σ^i is the (local) system state at site i and T_k is a transaction. Site i, if asked to vote on transaction T_k while at state σ^i , votes to commit iff $VOTE(\sigma^i,T_k)$ is true.

4.2 Extension of the Model

A distributed database design is a tripe (t,D,str), where $t \in \mathbb{N}$ is the number of sites, D is the set of data items and $str: D \rightarrow \{1,...,t\}$ is a function that maps each data item to the site at which it is stored. Note that making str a (single-valued) function commits us to databases with no

data replication, as each data item in D is stored at exactly one site. Throughout this section we keep the distributed database design fixed.

A global system state, σ , (for the given distributed database design) is a triple $\sigma = (S_{\sigma}, SWI_{\sigma}, L_{\sigma})$, where the three components have the same meaning as the components of a system state as defined in Section 2. Σ denotes the set of global system states.

A local system state at site i, σ^i , is a triple $\sigma^i = (S_{\sigma^i}, SWI_{\sigma^i}, L_{\sigma^i})$ where the three components are restricted to data items stored at site i. Thus, if we define $D^i = \{x \in D | str(x) = i\}$, $S_{\sigma^i} : D^i \to HU$, $SWI_{\sigma^i} \subseteq D^i \times HU \times T$ and L_{σ^i} is a log such that if $R_k[x] \in OP_{L_{\sigma^i}}$ or $W_k[x] \in OP_{L_{\sigma^i}}$, $x \in D^i$, Σ^i denotes the set of local system states at site i.

Given a global system state $\sigma \in \Sigma$, we may define a unique local system state $\sigma^i \in \Sigma^i$, for every site i as follows: $S_{\sigma^i} = S_{\sigma^i} | D^i$, $SWI_{\sigma^i} = \{(x, v, T_k) \in SWI_{\sigma^i} | x \in D^i\}, \text{ and } L_{\sigma^i} \text{ is the restriction of the partial order } L_{\sigma^i} \text{ to the domain } OP_{L_{\sigma^i}} = \{a_k \in OP_{L_{\sigma^i}} | a_k \in \{C_k, A_k, R_k[x], W_k[x] | x \in D^i\}\}.$

Given a vector of system states $(\sigma^1, \dots, \sigma^t)$ where $\sigma^i \in \Sigma^i$, we may total define a global system state σ^i , as follows: $\sigma^i = \sigma^i$ and σ^i is any log with domain $\sigma^i = \sigma^i$ by $\sigma^i = \sigma^i$ and $\sigma^i = \sigma^i$ by $\sigma^i = \sigma^i$ and $\sigma^i = \sigma^i$ by $\sigma^i = \sigma^i$ for every $1 \leq i \leq t$. Note that this does not uniquely define $\sigma^i = \sigma^i$ may be several logs with the wanted property.

According to one school of distributed database researchers it is the job of the transaction manager (not of the scheduler or data manager) to ensure that, in replicated databases, all copies of a data item be kept consistent. This view is explicitly taken, for example, in [KP81]. As ours is essentially a model at the data manager level, this view would justify our assumption. We hope to explore the complications of data replication in in another paper.

Throughout this section we adopt the convention that local system states σ^i and global system state σ are related as above

A distributed algorithm (for processing database operations) is a pair $(\lambda_{\mathcal{D}}, \gamma_{\mathcal{D}})$ where $\lambda_{\mathcal{D}} \colon \Sigma \to 2^{\Sigma}$ and $\gamma_{\mathcal{D}} \colon \Sigma \to 2^{\Sigma}$. Intuitively the distributed algorithm changes the global system state to a new one by one application of either $\lambda_{\mathcal{D}}$ or $\gamma_{\mathcal{D}}$ to the "current" system state. The reason we broke the state transition function into two components is that $\lambda_{\mathcal{D}}$ corresponds to changes of the global state incurred by local processing at the sites, while $\gamma_{\mathcal{D}}$ corresponds to changes of the global state incurred by the coordinated activity of aborting or committing a transaction—i.e. $\gamma_{\mathcal{D}}$ simulates (in a very crude way) the activity of atomic commitment.

More formally, we require of a distributed algorithm $(\lambda_{\mathcal{D}}, \gamma_{\mathcal{D}})$ that:

- (DA1) If $\sigma' \in \lambda_{\mathcal{D}}(\sigma)$ then for all transactions T_k , $C_k \in OP_{L_{\sigma'}} \longleftrightarrow C_k \in OP_{L_{\sigma'}}$ and $A_k \in OP_{L_{\sigma'}} \longleftrightarrow A_k \in OP_{L_{\sigma'}}$, and
- (DA2) If $\sigma' \in \gamma_{\mathcal{D}}(\sigma)$ then for some transaction T_k , either $L_{\sigma'} = L_{\sigma} \circ C_k$ or $L_{\sigma'} = L_{\sigma} \circ A_k$.

A global history is a sequence $\sigma_1 \sigma_2 \dots \sigma_n \in \Gamma^*$. A global history $\sigma_1 \sigma_2 \dots \sigma_n$ is (λ_D, γ_D) -compatible iff $\sigma_{i+1} \in \lambda_D(\sigma_i)$ or $\sigma_{i+1} \in \gamma_D(\sigma_i)$ for $1 \le i \le n$. For a log L we define $h_D(L) \stackrel{\Delta}{=} \{\alpha = \sigma_0 \sigma_1 \dots \sigma_n | \alpha \text{ is } (\lambda_D, \gamma_D)$ -compatible and $L_{\sigma_n} = L \}$. We may define resilient global system states, and the concepts of the last committed writer of a data item x in a global system state σ , denoted $lcw_{\sigma}(x)$, and of the committed database state in a global system state σ , denoted CS_{σ} , exactly as in Section 2. We shall not duplicate these definitions here. \dagger

Note, however, that for these definitions to be immediately transferable in our new setting, it is crucial that there be no replicated data items. In fact, one concern in studying data replication will be to find conditions on the algorithms such that low and CS are always well-defined.

4.3 Building Reliable Distributed Algorithms

We are going to build reliable distributed algorithms on the basis of correct and well-defined centralized algorithms, as defined in Section 2 (and examples of which were analyzed in Section 3). Intuitively, each site uses a (centralized) algorithm to process the data operations submitted to it. Eventually transactions must become globally committed or globally aborted.

Let $\mu_{\mbox{$A$}_{\mbox{$i$}}}$ be the (centralized) algorithm on which is based the data processing algorithm of site i. We assume that $\mu_{\mbox{$A$}_{\mbox{$i$}}}$ is correct (see Definition 2.5) and well-formed (see Definition 2.8).

Let $\sigma^i \in \Sigma^i$ and T_k be a transaction. We define, for each site i, a predicate VOTE, as follows:

Definition 4.1.
$$VOTE_{i}(\sigma^{i},T_{k}) = \exists \sigma^{i} \in \Sigma^{i}(\sigma^{i} \in \mu_{A_{i}}(\sigma^{i}) \land L_{\sigma^{i}} = L_{\sigma^{i}} \circ C_{k}).$$

Note that, by the fact that $\mu_{A_{\hat{i}}}$ is well-formed, for any transaction T_k there exist states σ^i that render $\text{VOTE}_i(\sigma^i,T_k)$ true.

The predicate $VOTE_i$ is used by site i to decide how to vote on a transaction T_k if it is asked to participate in an atomic commitment protocol. If site i is at state σ^i when that happens, it votes to commit T_k , if $VOTE_i(\sigma^i, T_k)$ is true, and to abort T_k , otherwise. Intuitively, $VOTE_i(\sigma^i, T_k)$ is true if the centralized algorithm μ_A could commit T_k in the next state transition from σ^i .

We may now define functions $\lambda_{\mathcal{D}} \colon \Sigma \to 2^{\Sigma}$ and $\gamma_{\mathcal{D}} \colon \Sigma \to 2^{\Sigma}$ as follows:

<u>Definition 4.2</u>. $\sigma' \in \lambda_{\mathcal{D}}(\sigma)$ iff

$$[\sigma^{,i} \in \mu_{A_{\underline{i}}}(\sigma^{\underline{i}}) \land \forall T_{\underline{k}} [(C_{\underline{k}} \in OP_{\underline{L}_{G}}, \longleftrightarrow C_{\underline{k}} \in OP_{\underline{L}_{G}}) \land (A_{\underline{k}} \in OP_{\underline{L}_{G}}, \longleftrightarrow A_{\underline{k}} \in OP_{\underline{L}_{G}})])$$

$$\lor \sigma^{,i} = \sigma^{i}, \quad \text{for } 1 \leq i \leq t.$$

 $\sigma' \in \gamma_{\mathcal{D}}(\sigma)$ iff

$$\sigma^{,i} \in \mu_{A_{i}}(\sigma^{i}) \land \exists T_{k} [(\land VOTE_{i}(\sigma^{i}, T_{k}) \land L_{\sigma}, = L_{\sigma} \circ C_{k})$$

$$\vee (\sim \bigwedge_{i=1}^{t} VOTE_{i}(\sigma^{i}, T_{k}) \land L_{\sigma}, = L_{\sigma} \circ A_{k})], \text{ for all } 1 \leq i \leq k.$$

$$\sigma^{,i} \in \mu_{A_{i}}(\sigma^{i}) \land \exists T_{i}(\sigma^{i}, T_{k}) \land L_{\sigma}, = L_{\sigma} \circ A_{k})], \text{ for all } 1 \leq i \leq k.$$

Informally, this definition says that a global system state transition by λ_D occurs if one or more sites change their local system state due to local processing. A transition by γ_D occurs when a transaction is globally committed or aborted. If every site i is at a local system state σ^i at which it can commit transaction T_k in the next step by its local (centralized) algorithm (i.e., $\text{VOTE}_i(\sigma^i, T_k)$ is true for $1 \le i \le t$), then T_k is globally committed. Otherwise T_k is globally aborted.

Now we must show that a distributed algorithm $(\lambda_{\mathcal{D}}, \gamma_{\mathcal{D}})$, constructed as above is *correct*. Let $\sigma_0 \in \Sigma$ be the initial global system state and $\sigma_0^i \in \Sigma^i$ be the initial local system state at site i. σ_0 and σ_0^i are resilient (see Lemma 2.4).

THEOREM 4.3. Let $(\lambda_{\mathcal{D}}, \gamma_{\mathcal{D}})$ be constructed as in Definition 4.2 and let $\sigma_0 \sigma_1 \dots \sigma_n$ be a $(\lambda_{\mathcal{D}}, \gamma_{\mathcal{D}})$ -compatible history. Then σ_n is resilient.

<u>Proof.</u> We show, by induction on i, that σ_i is resilient, $0 \le i \le n$.

<u>Basis:</u> As we remarked, σ_0 is resilient.

Induction Step: Suppose σ_j is resilient for some $1 \le j \le n$. We'll show that σ_{j+1} is also reslient. By inspection of Definition 4.2 it is easy to see that $\lim_{\sigma_{j+1}} \sup_{\sigma_j} \sup_{\sigma_{j+1}} \sup_{\sigma_j} \sup_{\sigma_{j+1}} \sup_{\sigma_j} \sup_{\sigma_{j+1}} \sup_{\sigma_j} \sup_{\sigma_j}$

We can also prove the analogue of Theorem 2.13 for a distributed database. We shall state the theorem in this new setting but will not give the proof as it is similar to that of 2.13.

THEOREM 4.4. Let $(\lambda_{\mathcal{D}}, \gamma_{\mathcal{D}})$ be a distriluted algorithm defined as in 4.2, $\mathbf{L} \in \mathbb{RC}$, $\sigma_0 \sigma_1 \dots \sigma_n \in \mathbf{h}_{\mathcal{D}}(\mathbf{L})$ and $\sigma_0 \tau_1 \dots \tau_m \in \mathbf{h}_{\mathcal{D}}(\pi_{\mathsf{Com}(\mathbf{L})}(\mathbf{L}))$. Then $\mathsf{CS}_{\sigma_n} = \mathsf{CS}_{\tau_m}$.

FINAL REMARKS

In this paper we developed a model of a database system for the purpose of studying formally the reliable processing of operations in such a system. Our motivation for doing so is that in most of the work on database reliability we are familiar with, notions such as "correct algorithm", "resilient database", "transaction redo", "transaction undo" etc. are used loosely and defined in the limited context of particular database systems under development.

We were able to use our model to define such concepts formally in a way that, we hope, makes intuitive sense. We believe that our model provides a reasonable abstraction in which algorithms for processing database operations can be described and their reliability properties studied. Partial evidence supporting our belief can be found in Section 3 of the present paper, where we state four fundamental algorithms and rigorously prove their reliability-related properties.

Lest we be mistaken for proposing our model as the database reliability panacea let us point out that it is ineffective—if not misleading—for analyzing the performance of the algorithms that can be described in its terms. For example, from the redo/undo properties of the four algorithms described in Section 3 one might fallaciously conclude that algorithm IV has superior performance than either algorithm II or algorithm III, since algorithm IV never requires either redo or undo while the other two require one of them. By the same token one might also fallaciously conclude that algorithms II and III are both superior to algorithm I, which requires both redo and undo.

The reason such conclusions are unwarranted is that the data structures used in known implementations of algorithm I ([Lo77], [V78], [HR79], [BGH82]) may incur significant overhead. For example, the one implementation that is known to achieve the properties of Lorie's algorithm tends, to cause physical separation of logically related disk pages, thus adversely affecting disk seek time. Our point is that models much more detailed than the one developed in this paper seem to be necessary before we are able to substantiate claims concerning the performance properties of different algorithms for processing database operations.

One direction in which our model can be extended is toward replicated distributed databases. One would hopefully be able to define the properties that distributed algorithms for processing transactions must satisfy in order for the different copies of each data item to be consistent. One would also hope to relate such properties to log-theoretic properties of data replication, as in [ABG82] and [BG82].

Another hopeful direction is to extend our model for the purposes of studying atomic commitment algorithms. The idea is to equip local system states with two more components: one for messages to be delivered to other sites and another for messages received. There will be "local transitions" which can modify the database state, S, the stable write information, SWI, and the log, L, components of a local system state; and "global (or network) transitions" which "deliver" messages from one site to another by modifying the two new components. We must then define various algorithms as transition functions of this kind and also different classes of faults. One would hope that this will lead to rigorous proofs of resiliency properties of various atomic commitment algorithms (described

as state transition functions) with respect to the different classes of faults. At the time of this writing, however, we have not yet experimented with this idea enough to have concluded whether or not it is a better way for studying atomic commitment protocols than other formalisms, such as the finite state automata-based formalism of Skeen [S82].

REFERENCES

- [ABG82] R. Attar, P.A. Bernstein, and N. Goodman, "Site initialization, recovery and backup in a distributed database system,"

 Proceedings of the 1982 Berkeley Workshop on Distributed Databases and Computer Networks, 1982 (also issued as Aiken Computation Laboratory TR-13-81, Harvard University).
- [BG82] P.A. Bernstein, and N. Goodman, "Multiversion concurrency control-theory and algorithms," Proceedings of the ACM SIGACT-SIGOPS Conference on Principles of Distributed Computation, August 1982, Ottawa (also issued as Aiken Computation Laboratory TR-20-82, Harvard University).
- [BGH82] P.A. Bernstein, N. Goodman, and V. Hadzilacos, "Recovery algorithms for database systems," to appear, *Proceedings of 1983 IFIP Conference*.
- [D82] D.J. Dubourdieu, "Implementation of distributed transactions,"

 Proceedings of the 1982 Berkeley Workshop on Distributed Data

 Management and Computer Networks, 1982, pp. 81-94.
- [G78] J.N. Gray, "Notes on database operating systems," in Operating Systems: An Advanced Course, Springer-Verlag, 1978.
- [H82] V. Hadzilacos, "Formalizing recovery in database systems," unpublished memorandum, Aiken Computation Laboratory, Harvard University, May 1982.
- [HR79] T. Harder, and A. Reuter, "Optimization of logging and recovery in a database system," in *Database Architecture*, Bacchi and Nijssen (eds.), North-Holland, 1979, pp. 151-168.
- [HR82] T. Harder, and A. Reuter, "Principles of transaction oriented database recovery--a taxonomy," Universität Kaiserslautern, FRG, Technical Report 50/82, April 1982.
- [HS80] M. Hammer, and D.W. Shipman, "Reliability mechanism for SDD-1:

 A system for distributed databases," ACM Transactions on Database Systems 5:4, December 1980, pp. 431-466.
- [KP81] P.C. Kanellakis, and C.H. Papadimitriou, "The complexity of distributed concurrency control," Proceedings of the IEEE 22nd Annual Symposium on Foundations of Computer Science, October 1981, pp. 185-197.
- [Li79] B.G. Lindsay, et al., "Notes on distributed databases," IBM Research Report, No. RJ2571, July 1979.

- [Lo77] R.A. Lorie, "Physical integrity in a large segmented database,"

 ACM Transactions on Database Systems 2:1, March 1977, pp. 91-104.
- [LS76] B. Lampson, and H. Sturgis, "Crash recovery in a distributed data storage system," unpublished memorandum, Xerox PARC, 1976.
- [Ly82] N.A. Lynch, "Concurrency control for resilient nested transactions," unpublished manuscript, Laboratory for Computer Science, MIT, May 1982.
- [ML80] D.A. Merascé, and O.E. Landes, "On the design of a reliable storage component for distributed database management systems," Proceedings of the Conference on Very Large Database (ULDB), Montreal, 1980, pp. 365-375.
- [R75] R.L. Rappaport, "File structure design to facilitate on-line instantaneous updating," *Proceedings of the 1975 SIGMOD Conference*, pp. 1-14.
- [S81] D. Skeen, "Nonblocking commit protocols," Proceedings of the 1981 ACM-SIGMOD International Conference on Management of Data, April 1981, pp. 133-142.
- [S82] D. Skeen, "Crash recovery in a distributed data base system," Ph.D. Dissertation, Department of Electrical Engineering and Computer Science, University of California at Berkeley, 1982.
- [V78] J.S.M. Verhofstad, "Recovery techniques for database systems," ACM Computing Surveys 10:2, June 1978, pp. 167-195.

SECTION VIII

SITE INITIALIZATION, RECOVERY AND BACK-UP IN A DISTRIBUTED DATABASE SYSTEM

Rony Attar

Philip A. Bernstein

Nathan Goodman

^{*}Published in the Proceedings of the Sixth Berkeley Workshop on Distributed Data Management and Computer Networks, Asilomar, Feb. 1982.

ABSTRACT

Site initialization is the problem of integrating a new site into a running distributed database system (DDBS). Site recovery is the problem of integrating an old site into a DDBS when the site recovers from failure. Site backup is the problem of creating a static backup copy of a database for archival or query purposes. We present an algorithm that solves the site initialization problem. By modifying the algorithm slightly, we get solutions to the other two problems as well.

Our algorithm exploits the fact that a correct DDBS must run a serializable concurrency control algorithm. Our algorithm relies on the concurrency control algorithm to handle all inter-site synchronization.

1. THE SITE INITIALIZATION PROBLEM

Site initialisation is the problem of integrating a new site into a distributed database system (DDBS). The goal is to make the new site look like all other sites. In particular, transactions must be able to access data at the new site in the same way as they access data at all other sites. The main problem is to bring the database at the new site up-to-date relative to the rest of the system. The problem is caused by replicated data: if the new site stores datum X and there are copies of X elsewhere in the system, then the value of X at the new site must agree with its value in the rest of the system. There is a simple brute force solution to the problem: just turn off the DDBS, wait for all activity to subside, and then load the new site's database in bulk. Our solution is almost as simple as this, but far more practical.

Our algorithm exploits the fact that a correct DDBS must run a serializable concurrence control algorithm (cf. [BG]). Concurrency control is the activity of coordinating transactions that access a database concurrently. The goal is to prevent concurrent transactions from interfering with each other. This goal is usually formalized by the concept of serializability (e.g. [BSW, EGLT, Pa, SLR, Th]): an execution of transactions is serializable if it is equivalent to an execution in which transactions execute serially, one after the other with no concurrency. Many algorithms are known for attaining this goal, e.g. two phase locking and timestamp ordering.

As we will see, the site initialization problem can be neatly framed in terms of merializable executions. Once stated in these terms, a simple solution will stare us in the face. All we have to do is:

- (1) turn on the concurrency control algorithm at the new site;
- (2) tell all other sites to begin updating the replicated data at the new site; and

(3) not let any transaction read a datum X at the new site until X has been updated at least once.

These three steps are a sketch of our algorithm. The rest of the paper fills in the details, and explains why the algorithm works. We also show how to use the algorithm to solve site recovery and backup problems.

2. BASIC CONCEPTS

A distributed database system (DDBS) is a set of sites interconnected by a network. Each site runs two software modules: a transaction manager (TM), which supervises the execution of transactions; and a data manager (DM), which processes read and write operations on data stored at the site.

A logical database is a set of logical data items, denoted X,Y,Z. A copy of a logical data item stored at a site is called a physical data item. We use x_1, \ldots, x_m to denote the physical copies of X.

A transaction is a program that accesses the database by issuing READ and WRITE operations on logical data items. For notational convenience we assume that a transaction issues all of its READ's before any of its WRITE's.

Each transaction's execution is supervised by one TM. When a transaction issues an operation READ(X), its TM selects a copy of X, say x_i , and issues an operation read(x_i) to the DM that manages x_i . (We use upper case for logical operations and lower case for physical ones.) When a transaction issues an operation WRITE(X), its TM issues an operation write(x_i) for every copy x_i of X.

The logical data items that a transaction reads(resp.writes) is called the transaction's readset (resp.writeset).

We mathematically model executions of transactions in a DDBS by a log.

A log describes the order in which read and write operations are processed by

DM's. Formally, a *log* is a partial order* of read and write operations.

For example,

$$L_{1} = w_{0}[x_{1}, x_{2}, y_{1}, z_{1}] \xrightarrow{r_{2}[x_{1}] \longrightarrow w_{2}[x_{1}, x_{2}] \longrightarrow r_{3}[x_{1}] \longrightarrow w_{3}[z_{1}]}$$

$$L_{1} = w_{0}[x_{1}, x_{2}, y_{1}, z_{1}] \xrightarrow{r_{1}[y_{1}] \longrightarrow r_{1}[x_{2}] \longrightarrow w_{1}[x_{1}, x_{2}, y_{1}]}$$

is a log. Notationally, $r_i[x_j]$ (resp. $w_i[x_j]$) denotes the execution of a read (resp. write) operation by transaction i on data item x_i . The arrows indicate the partial order, which represents the order in which operations were executed. So, in L_1 , $w_0[x_1,x_2,y_1,z_1]$ executed before any other operation; $r_2[x_1]$ executed before $w_2[x_1,x_2]$ and $r_1[x_2]$, but it executed concurrently with $r_1[y_1]$; and so forth.

We place one more constraint on the allowed form of logs: for each physical data item $\mathbf{x_i}$, all operations on $\mathbf{x_i}$ must be totally ordered.**

That is, for each $\mathbf{x_i}$, we know the exact order in which operations on $\mathbf{x_i}$ occurred. We often relax this constraint for read operations, since the order of read's is unimportant anyway.

^{*}A partial order is a binary relation, \leq , that is reflexive (a \leq a), antisymmetrical (a \leq b and a \leq b implies a=b), and transitive (a \leq b and b \leq c implies a \leq c). Traditionally, a distributed execution is modelled as a set of sequential logs, one per DM [BG]. We prefer using partial orders because they allow operations from different DM's to be ordered and they do not require ordering unrelated operations that can be executed concurrently at the same DM.

^{**}A total order is a partial order in which every pair of elements are related (i.e., a≤b or b≤a). A total order is the same as a sequence.

Two logs are equivalent if they represent executions that produce the same final database state, and if each transaction performs the same computation in both executions. The following proposition states a well-known, and very useful, characterization of log equivalence. We need one more definition first. Two operations conflict if they operate on the same physical data item and one is a write.

Proposition 1 Two logs are equivalent if they contain the same operations, and every pair of conflicting operations appear in the same order in both logs.

3. CORRECTNESS CONCEPTS

The correctness of any system must be defined relative to user's expectations. Intuitively, a system is correct if it does what users want it to do. We assume that users expect a DDBS to behave like a serial transaction processor; that is, users expect the DDBS to behave as if it were processing transactions one at a time, against the logical, non-replicated database. (This assumption is adopted almost uniformly in the literature.) A DDBS is correct if it behaves like a serial transaction processor in this sense.

In this section we analyze DDBS correctness using the basic concepts of Section 2.

A serial log is a total order of operations such that for every pair of transactions, all operations of one transaction precede each operation of the other. For example,

$$\xrightarrow{r_1[y_1] \longrightarrow r_1[x_2] \longrightarrow w_1[x_1, x_2, y_1]}$$

is a serial log.

Consider any read operation in a serial log, e.g. $r_2[x_1]$ above. This operation is said to be read-from the nearest write operation before it that writes into its argument. E.g. in L_2 $r_2[x_1]$ reads-from $w_0[x_1,x_2,y_1,z_1]$, while $r_3[x_1]$ and $r_1[x_2]$ read-from $w_2[x_1,x_2]$. Similarly, transaction T_1 reads- x_k -from T_1 if T_1 indeed reads x_k , T_1 writes x_k , and $r_1[x_k]$ reads-from $w_1[x_k]$. E.g. in L_2 , L_2 reads- x_1 -from L_3 .

A serial log is one-copy equivalent (or simply 1-serial) if for each transaction T_i , and for each x_k that T_i reads, T_i reads- x_k -from the last transaction before T_i that writes into any copy of X.

The reader can verify that L_2 is 1-serial. However, if we change $w_2[x_1,x_2]$ to $w_2[x_2]$, the resulting log is not 1-serial.

$$\xrightarrow{r_1[y_1] \longrightarrow r_1[x_2] \longrightarrow w_1[x_1,x_2,y_1]}$$

 L_3 is not 1-serial, because T_3 reads- x_1 -from T_0 , which is not the last transaction before T_3 that wrote into any copy of X.

A 1-serial log represents a serial execution of transactions in which the replicated copies of each data item behave like a single logical data item.

Therefore, every 1-serial log is correct in the sense defined at the beginning of this section.

A log is serializable (SR) if it is equivalent to a serial log. A log is 1-serializable (1-SR) if it is equivalent to a 1-serial log. Since every 1-serial log is correct, and since every 1-SR log is equivalent to a 1-serial log, every 1-SR log is also correct.

We adopt 1-SR as our basic notion of correctness for the rest of this paper.

If sites are never added to a DDBS and sites never fail, attaining 1-SR is little more than a concurrency control problem. All we have to do is:

- (1) make sure that each transaction writes into all physical copies of its writeset, as described in Section 2; and
- (2) synchronize read and write operations using any serializable concurrency control algorithm, such as two-phase locking.

The following proposition states the correctness of these steps in terms of logs.

Proposition 2 A log is 1-SR if every transaction in the log writes into all copies of its writeset, and the log is SR.

4. SITE INITIALIZATION ALGORITHM

Suppose we have a DDBS that is running correctly -- i.e. its execution is 1-SR -- and suppose we add a new site to the system. We need to integrate the site into the DDBS in such a way that (1) all data at the site can eventually be read, and (2) the resulting execution remains 1-SR.

In this section we describe an algorithm that accomplishes this task. First, we use the concepts of Sections 2 and 3 to specify the kinds of executions permitted by our algorithm, and to argue that these executions are *correct* (i.e. satisfy requirements (1) and (2)). Then, we demonstrate an algorithm that meets the specification.

Specification and Correctness

The logs that our algorithm will allow satisfy the following properties.

- Al. Each transaction writes into all copies of its writeset, except possibly those copies at the new site.
- A2. By some time t, every data item at the new site has been written into at least once.

- A3. No transaction reads a data item at the new site until that data item has been written at least once.
- A4. The log is SR.
- A5. Let x_{new} be a copy of X at the new site, and let T_X be the first transaction that writes into x_{new} . By Al, T_X also writes into the other copies of X. Let T_X' be any transaction that writes into any copy of X after T_X wrote into that copy. Then T_X' must also write into x_{new} .

Stated a bit loosely, A5 simply means that once any transaction writes into x_{new} , all later transactions also write into x_{new} .

We now argue that if a log satisfies Al-A5 then it is correct.

- (1) A2 and A3 ensure that all data items at the new site are eventually readable, thereby attaining the first correctness requirement.
- (2) It remains to prove that if log L satisfies Al-A5, then L is 1-SR. By A4, L is SR; let L_s be any serial log equivalent to L. Consider any reads-from relationship in L_s , e.g. T_i reads- x_k -from T_j . L_s looks like:

$$L_s = \dots \longrightarrow w_j[x_k] \longrightarrow r_i[x_k] \longrightarrow \dots$$

and we must show that no transactions between T_j and T_i in L_s writes into any copy of X. We will show this by proving that every transaction that follows T_i and updates any copy of X, also writes into x_k .

Let T_{ℓ} be any transaction that follows T_{j} and updates X. If x_{k} is not a "new" data item, then T_{ℓ} writes into x_{k} by Al. Now suppose x_{k} is "new". By Al and Proposition 1, T_{ℓ} follows T_{j} in L, and so T_{ℓ} writes into x_{k} by A5. In either case, since T_{ℓ} writes into x_{k} , and since T_{i} reads- x_{k} -from T_{j} (and not from T_{ℓ}), T_{ℓ} cannot come between T_{j} and T_{i} . Q.E.D.

Algorithm

Rules Al-A5 form a blueprint for a simple site initialization algorithm.

Let us see how these rules can be attained algorithmically.

Al and A3 are trivial to implement. A4 is merely concurrency control.

Any serializable concurrency control algorithm can be used. The remaining rules can be implemented as follows:

- A2. For each logical data item X stored at the new site, run a copier transaction that reads a copy of X at an old site and writes that value into the new copy. (I.e. there is one copier transaction per X.) Copiers must be synchronized by the concurrency control algorithms exactly like all other transactions.
- A5. For each logical data item X stored at the new site, designate a guardian copy x of X at some old site. Beginning at some (arbitrary) time t after the new site is added, the site holding x alerts all transactions that update x to write into the new copy of X also. No transaction updates a data item at the new site unless told to do so by its guardian.

This description of our algorithm may seem too abstract, mainly because we have not pinned down the concurrency control algorithm. For definiteness, let us see how the algorithm works in conjunction with two-phase locking.

These five rules constitute our proposed site initialization algorithm.

We begin by reviewing the basic two-phase locking algorithm.

Associated with each physical data item is a set of locks. At any time, the set of locks on a physical data item may contain no locks, one write lock, or a set of read locks.

Suppose x_i is stored at DM_i . Before processing read(x_i) on behalf of transaction T_i , DM_i must set a read lock on x_i for T_i . Before processing

write (x_i) on behalf of T_j , DM_i must set a write lock on x_i for T_j .

If DM_i cannot set a lock for an operation, it delays the operation until the lock can be set.* When a transaction terminates, all of its locks are released.

Now let us see how to add a new site to the system. Suppose sites 1,2,...,n-1 are running properly and we wish to add site n. Site n begins the process by sending an "I'm up" message to the DM's at sites 1,2,...,n-1.

Suppose the DM at site i, DM_i , receives an "I'm up" message from site n. From then on, for each guardian copy x_g at DM_i , when DM_i processes a write (x_g) , it tells the TM that issued the write to also issue a write (x_n) , where x_n is the copy of X at the new site. The DM also instructs its local TM to execute copier transactions for each of its guardian copies. The copier for x_g must obtain a read lock on x_g and a write lock on x_n ; i.e. it must be synchronized like any other transaction.

DM uses the same two-phase locking algorithm as every other DM. However, it refuses to process a read(\mathbf{x}_n) until \mathbf{x}_n has been updated at least once.

For each logical data item X, a TM issues a write(x_n) on behalf of a transaction that updates X, if and only if one of its writes on x_g has been acknowledged by a message telling it to do so. The TM must not update x_g until this point in time.

^{*}Since operations can be delayed while waiting for locks, deadlock is possible. Deadlocks can be resolved by any of the standard techniques in [BG].

5. SITE RECOVERY

Site recovery is the problem of integrating a site into a DDBS when the site recovers from failure. As for the site initialization problem, the goal is to make the recovered site look like all other sites. Once again, the main problem is to bring the database at the recovered site up-to-date relative to the rest of the DDBS.

Site recovery is obviously an important problem, but it has received little attention in the literature. One early paper on DDBS reliability [AD], which mainly studies reliable concurrency control algorithms, disposes of site recovery with these few words:

How the new host is brought up to date depends on the application. It may be done by transferring to that host the journal of all updates since the host went down. It may require transferring the database. [AD, p. 568].

Other related work includes [HS, LS, LSP, MPM, Th, Sk, SS]. Some of these papers [MPM, Th] are like [AD] in that they mainly study reliable concurrency control. (A piece of the algorithm in [MPM] is called Single Node Recovery. But the algorithm only recovers the concurrency control algorithm at the site, not the database.) Other papers study atomic commitment [HS, LS, Sk, SS], site monitoring to keep track of which sites are up [HS], and other distributed decision problems [LSP]. Again, site recovery in our sense is not studied.

One paper that does treat site recovery is [HS]. The solution is based on the concept of Reliable Network (Relnet), a virtual machine that guarentees reliable message delivery despite site failures. The Relnet is intended to be a very general facility suitable for many kinds of distributed systems. Because of this generality, the mechanism is rather complex.

Our approach to site recovery is narrower (and, we hope, simpler) than the Relnet approach. We are not trying to attain reliability for arbitrary distributed systems; nor are we trying to solve all DDBS reliability problems. Our goal is simply to integrate the database at a recovered site into a running DDBS.

Evidently, site recovery and site initialization are almost identical problems, and the algorithm of Section 4 can be directly applied here.

There is one major caveat: our algorithm says nothing about multiple failures. We believe the algorithm can be generalized to handle multiple failures, but offer no hard evidence in this respect. Despite this caveat, the algorithm of Section 4 solves a big piece of the site recovery problem.

An Optimization

When using the initialization algorithm for site recovery, an important optimization is possible. It is not necessary to fire up copier transactions for all X in the logical database. Suppose we are recovering site f.

Only those X that were updated after site f failed need to be recovered.

Any X that was not updated while f was down still has the correct value at f when the site recovers. If a spool (or journal) of all writes to site f is maintained while f is down, as in SDD-1 [HS], then when f recovers the following processing can be done. Scan the spool to produce a list of data items that were written while f was down. All data items not on the list can be immediately marked as readable at DM_f. Copiers are executed only for data items on the list.

Notice that we are not proposing that spooled write operations be processed in FIFO order, as in SDD-1. If X was written several times while f was down, only the last value should be sent to f. If earlier values are sent, the algorithm will not work correctly.

6. SITE BACKUP

A backup database is a static copy of the database that is consistent but potentially out-of-date. One use of backup databases is to speed up the processing of queries. By reading the static backup, a query does not interfere with updates, and so will not be delayed or restarted for concurrency control

reasons. The cost is that it may read out-of-date data. Backup databases are also useful for archiving data.

Creating a backup database is similar to initializing a new site or recovering a failed site -- similar enough that we can use our initialization algorithm to do most of the job.

We begin by pretending that the backup database is a new site being added to the DDBS. We run the initialization algorithm to bring the backup database up-to-date, until all data items in the backup have been written at least once.

Now we must freeze the backup, by shutting off writes to it. However, we must shut off writes carefully, so that the backup is frozen in a consistent state.

We can do this simply by running a query that (conceptually) reads the entire backup database, and by ensuring that no data item is written once the query has read it. This freezes the backup copy in the state read by the query. Since the query is synchronized by a serializable concurrency control algorithm, the frozen state is consistent.

For example, suppose we use the two-phase locking initialization algorithm of Section 4. When all backup data items have been written at least once, we run a query that sets a read lock on every backup data item. (The query may deadlock while trying to obtain its locks, and so may need to be aborted and restarted.) When all backup data items are locked, we shut off updating by refusing to process any more writes against the backup. The resulting backup database is consistent and can be correctly queried without synchronization.

One problem with this algorithm is that the "shut-off" query may deadlock repeatedly and never finish. This problem can be fixed as follows. Once the query begins, the backup should refuse to grant any write lock requests from transactions that have not already set a lock on some backup copy. These requests are simply blocked, and the transactions delayed, until the query manages to get all of its locks. Then a very counterintuitive event happens -- the lock

requests are unblocked, but since the backup is now frozen, the transactions no longer need the locks! (It does not work to unblock the transactions earlier.)

7. CONCLUSION

We have presented an algorithm that can be used to initialize a database at a new site in a DDBS, to recover a database at a formerly failed site, or to create a consistent, static backup database. The algorithm is simple, yet introduces little overhead beyond what is normally needed for concurrency control. We therefore believe it is a practical solution to all three problems.

The methodology that we used to describe our algorithm is also interesting, we believe. First, we defined correctness, i.e. what it means for an algorithm to correctly solve the problem; this definition was stated in terms of executions (i.e. logs). Second, we specified the kinds of logs that our algorithms would allow, and proved that every allowable log is correct. Third, we described an abstract algorithm that meets the specification. Finally, we gave a concrete implementation of the abstract algorithm. These four steps,

- (i) defining correct logs,
- (ii) specifying an allowable subset of the correct logs,
- (iii) designing an abstract algorithm that produces allowable logs,
- (iv) engineering a concrete implementation of the abstract algorithm, helps structure the problem and the search for solution.

One benefit is that we can engineer new concrete algorithms for specific systems or problems just by redoing step (iv). For example, the concrete implementation of the backup algorithm in Section 6, may have bad performance characteristics: By locking the entire backup database, the "shut-off" query interferes with many updates. This performance problem is not inherent in the abstract algorithm; it is just an artifact of the concrete implementation we gave. A better implementation would use a concurrency control algorithm for

the backup in which queries and updates interfere less. Multiversion concurrency control algorithms [BHR, Re, SR] are likely candidates for this role. Engineering a backup algorithm that uses multiversion concurrency control is by no means a trivial task. But structuring the problem as we have done, the designer does not have to start from scratch.

REFERENCES

- [AD] Alsberg, P.A., J.D. Day, "A Principle for Resilient Sharing of Distributed Resources," Proc. 2nd Intl. Conf. Software Eng., Oct. 1976.
- [BG] Bernstein, P.A., and N. Goodman, "Concurrency Control in Distributed Database Systems," ACN Computing Surveys, Vol. 13, No. 2, (June 1981).
- [BHR] Bayer, R., H. Heller, and A. Reiser, "Parallelism and Recovery in Database Systems," ACM Trans. on Database Syst., Vol. 5, No. 2 (June 1980), pp. 139-156.
- [BSW] Bernstein, P.A., D.W. Shipman, and W.S. Wong, "Formal Aspects of Serializability in Database Concurrency Control," *IEEE Trans. Softw. Eng.*, Vol. SE-5, No. 3 (May 1979).
- [EGLT] Eswaran, K.P., J.N. Gray, R.A. Lorie, and I.L. Traiger, "The Notions of Consistency and Predicate Locks in a Database System." Commun. ACM Vol. 19, No. 11, (Nov. 1976), pp. 624-633.
- [HS] Hammer, M.M., and D.W. Shipman, "Reliability Mechanisms for SDD-1:
 A System for Distributed Databases," ACM Trans. Database Syst. Vol. 5,
 No. 4 (Dec. 1980), 431-466.
- [LS] Lampson, B., and H. Sturgis, "Crash Recovery in a Distributed Data Storage System," Tech. Rep., Computer Science Lab., Xerox Palo Alto Research Center, Palo Alto, CA, 1976.
- [LSP] Lamport, L., R. Shostak, and M. Pease, "The Byzantine Generals Problem," Tech. Rep., SRI International, March 1980.
- [MPM] Menasce, D.A., G.J. Popek, and R.R. Muntz, "A Locking Protocol for Resource Coordination in Distributed Databases," ACM Trans. Database Syst. Vol. 5, No. 2, (June 1980), pp. 103-138.
- [Pa] Papadimitriou, C.H., "Serializability of Concurrent Updates," J. ACM Vol. 26, No. 4 (Oct. 1979), pp. 631-653.
- [Re] Reed, D.P., "Naming and Synchronization in a Decentralized Computer System", Ph.D. dissertation, Dept. of Electrical Engineering, M.I.T., Cambridge, MA, Sept. 1978.
- [Sk] Skeen, Dale, "Nonblocking Commit Protocols," Proc. 1981 ACM-SIGMOD Conf. on Management of Data, ACM, N.Y., pp. 133-147.
- [SLR] Stearns, R.E., P.M. Lewis, II, and D.J. Rosenkrantz, "Concurrency Controls for Database Systems," in *Proc. 17th Symp. Foundations Computer Science (IEEE)*, 1976, pp. 19-32.
- [SR] Stearns, R.E., and D.J. Rosenkrantz, "Distributed Database Concurrency Controls Using Before-Values," in *Proc. 1981 ACM-SIGMOD Conf. on Management of Data*, ACM, N.Y., pp. 74-83.

- [SS] Skeen, D., and M. Stonebraker, "A Formal Model of Crash Recovery in a Distributed System", Proc. 5th Berkeley Conference on Distributed Data Management and Computer Networks, 1981, pp. 129-142.
- Thomas, R.H., "A Majority Consensus Approach to Concurrency Control for Multiple Copy Databases", ACM Trans. on Database Syst., Vol. 4, No. 2 (June 1979), pp. 180-209.

SECTION IX

AN ALGORITHM FOR MINIMIZING ROLL BACK COST*

Vassos Hadzilacos

^{*}Published in the Proceedings of the First ACM SIGACT-SIGMOD Symposium on Principles of Database Systems, Los Angeles, March 1982.

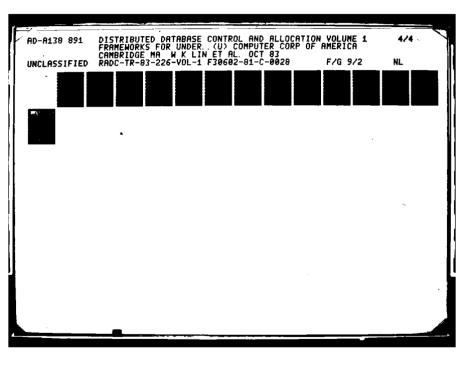
O. INTRODUCTION

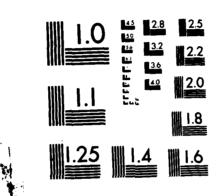
Most automatic crash recovery mechanisms for database systems are based on the concept of transaction commitment. Speaking very informally, when the system designates a transaction to be committed, it "promises" to install in the database all the updates effected by that transaction. Put another way, should a crash occur after a transaction has become committed, the transaction may not be restarted. If, however, a crash occurs before a transaction has become committed, that transaction must be restarted. Several mechanisms that achieve this behavior have been proposed by database system designers (e.g. [R 75], [Lo 77], [G 78], [Li 79]). In all these systems, when a transaction is restarted, it is "rolled back" all the way to the beginning. One exception to this rule is System-R which allows to roll an uncommitted transaction back to an earlier "savepoint", which is not necessarily its beginning [A 76].

In this paper we investigate the problem of finding the optimal set of savepoints (one per transaction) to which uncommitted transactions executing concurrently must be rolled back after a crash, so that the recovery cost is minimized.

This paper is organized as follows. Section 1 informally motivates the problem. In Section 2 we present a more formal model of transaction execution in terms of which our results are stated and proved. In Section 3 we give an algorithm for the problem under consideration and prove its correctness and optimality. In many environments, cascading restarts are considered intolerable, and sufficiently restrictive schedulers are used to prevent them. In such environments the problem dealt with in this paper has a trivial solution.

This issue is discussed in Section 4.





MICROCOPY RESOLUTION TEST CHART NATIONAL BUREAU OF STANDARDS-1963-A

1. THE PROBLEM

We consider a database system in which transactions operate on the database concurrently. The system periodically takes "transaction savepoints" for the transactions currently active. A transaction savepoint involves saving the current state of a transaction in non-volatile storage. What exactly constitutes the "current state" of a transaction depends on details that we do not care to consider here. Typically it would contain such information as the program counter, the values of all local variables created by the transaction so far, any locks held by the transaction, etc. We make no assumptions concerning the timing of the savepoints. They may be asynchronous (i.e. happening at different times for different transactions), or occurring with different frequencies for different transactions. They system may be taking savepoints at its convenience. Alternatively, the transactions themselves could specify when savepoints are to be taken. All the savepoints of a transaction must be kept until that transaction becomes committed. That is, a savepoint of a transaction must not "overwrite" a previous savepoint of that (or any other) transaction.

If a crash occurs, the recovery algorithm selects an appropriate set of savepoints, one per (uncommitted) transaction. Each (uncommitted) transaction is then rolled back to its corresponding savepoint, and processing continues from there, as if the crash had never occurred.

We are only concerned with "soft" crashes in this paper--i.e. crashes in which volatile storage is lost, but non-volatile storage is not affected.

We must, of course, "synchronize" the actual database with the state of affairs reflected by rolling back the uncommitted transactions. For example, all updates of any such transaction that occurred after the savepoint to which the transaction is rolled back must be "undone" from the database. This can be achieved if the database system maintains an audit trail. This technique is well-known (see, for example, [G 78], [Li 79]) and will not be addressed here. This question is dealt with in an environment akin to the one considered here in [F 81]. Also, the conditions under which rolling back transactions does not endanger the correctness of the database state are discussed in [H 81].

At this point it might appear sufficient to simply roll back each transaction to its last savepoint. Unfortunately, this naive approach could well lead to inconsistent database states. The fundamental reason is that operations on a database issued by different transactions are not independent of each other. Informally, we say that an operation a depends on operation b, if a was executed after b and the result of a would have been different if b had not been executed. For example, a Read(x) depends on the immediately previous Write(x), and a Lock(x) depends on the immediately preceding Unlock(x) (assuming exclusive locking). The exact nature of the dependence of operations on one another is not important here. We simply assume that some dependence relation is specified which correctly reflects the semantics of the various operations.

Consider now the following sequence of events:

- (1) savepoint of transaction T, is taken
- (2) operation a_i of T_i is executed
- (3) operation a of transaction T is executed
- (4) savepoint of T_{i} is taken
- (5) the system crashes.

Assume, further, that operation a_j depends on a_i . If we were to roll back each of T_i , T_j to their last savepoints, we would be restarting the system at a state in which operation a_j is executed (since it happened before T_j 's last savepoint), while the a_i (on whose execution a_j depends) is not executed (since it happened after T_i 's last savepoint). This is an inconsistent execution state, at least according to our informal notion of operation dependence.

This example illustrates that it is not always correct to roll back each (uncommitted) transaction to its last savepoint. Our task now is to provide

an algorithm for choosing a set of savepoints S*, one per uncommitted transaction, such that

- (i) the execution state reflected by rolling back each transaction to the corresponding savepoint is consistent, and
- (ii) S* is, in a sense, optimal.

These two conditions will be defined precisely in the next section.

2. FORMAL MODEL

Each transaction T, consists of a sequence of atomic steps or operations $a_{i1}, a_{i2}, \ldots, a_{i\lambda_i}$. For each T_i we also have a set of savepoints $SP_i = \{s_{i1}, s_{i2}, \dots, s_{im_i}\}$ and a mapping $\phi_i: SP_i + \{1, 2, \dots, \lambda_i\}$, such that $\phi_{i}(s_{ij}) < \phi_{i}(s_{ik})$, if $1 \le j < k \le m_{i}$. Intuitively, savepoint s_{ij} was taken after the execution of step a and before the execution of a_{i,k+1} where $k = \phi_i(s_{ij})$. A concurrent execution of T_1, \dots, T_n is a sequence consisting of the elements of $A = \{a_{ij} : 1 \le i \le n, 1 \le j \le \lambda_i\}$, respecting the order of atomic steps belonging to the same transaction. With respect to a concurrent execution E of $T_1, ..., T_n$ we are given a partial order $\rightarrow_F \subseteq A \times A$, where we require that if $a_{ij} \stackrel{+}{\rightarrow} a_{i'j'}$, then a_{ij} appears before $a_{i'j'}$, in E. $\div_{_{\rm E}}$ is called the dependence relation, and formalizes the intuitive notion of operation dependence discussed in the previous section: if $\mathbf{a} \xrightarrow{\mathbf{r}} \mathbf{b}$ then in execution E, step a "depends" on step b. Since we shall be dealing with a fixed execution E, the subscript will be omitted from the symbol " \rightarrow_{Σ} ". Let $s = \{s_{lt_1}, \dots, s_{nt_n}\}$ where, as usual, s_{it_4} is a savepoint of transaction T_i . We shall say that S is a consistent set of savepoints (with respect to an execution E), iff S satisfies the following condition:

(*) if $a_{ij} + a_{i'j}$, then either $j \le \phi_i(s_{it_i})$ or $j' > \phi_i(s_{i't_i})$.

Intuitively this means that if an operation $a_{i'j}$, depends on a_{ij} and we roll back each transaction T_i to the corresponding savepoint in S, we are restarting the system in a state where either a_{ij} has been executed $(j \le \phi_i(s_{i'j})$ or $a_{i'j}$, has not been executed $(j' > \phi_i, (s_{i't_{i'}}))$. As we argued informally in the previous section, unless this condition is satisfied, the state we are restarting our system from is inconsistent.

Now we need to define what is meant by an "optimal" set of savepoints. To do so, we define the class of "reasonable" cost functions. Let s_{it_i} , s_{ir_i} be savepoints of T_i . We say that $f(x_1, \ldots, x_n)$ is a reasonable cost function iff for all j, $1 \le j \le n$, if $t_i = r_i$ for $i \ne j$ and $t_j < r_j$, then $f(s_{1t_1}, \ldots, s_{nt_n}) > f(s_{1r_1}, \ldots, s_{nr_n})$. This captures the idea that the more we roll back, the greater the cost of recovery.

We say that a set of savepoints S* is optimal iff it minimizes a given reasonable cost function among all consistent sets of savepoints. Fortunately, the same S* minimizes all reasonable cost functions (Lemma 3).

The class of reasonable functions includes such obvious candidates for optimization as $f(s_{lt_1}, \ldots, s_{nt_n}) = \sum_{i=1}^n c(s_{it_i})$ and $f(s_{lt_1}, \ldots, s_{nt_n}) = \max_{1 \le i \le n} c(s_{it_i})$, where $c(s_{it_i})$ is the cost of restarting T_i from savepoint s_{it_i} , under the assumption that it costs more to restart a transaction from an earlier savepoint.

3. THE ALGORITHM AND THE PROOF

We now describe an algorithm for finding the optimal, consistent set of savepoints that minimizes any reasonable cost function.

Algorithm 1

<u>Input</u>: For each transaction T_i the set SP_i , the function ϕ_i : $SP_i + \{1,2,\ldots,\lambda_i\}$, and the dependence relation + on the set $A = \{a_{ij}: 1 \le i \le n, 1 \le j \le \lambda_i\}$.

Output: A consistent set of savepoints S* minimizing any reasonable cost function.

Method:

Step 0: Construct a digraph G = (N,E) with node set $N = \{s_{ij} : 1 \le i \le n, 1 \le j \le m_i\} \cup \{s_{i0} : 1 \le i \le n\}$, where $\phi_i(s_{i0}) = 0$ by fiat, and edge set $E = \{(s_{ij}, s_{i'j'}) : \text{ there are steps } a_{ip'}, a_{i'p'}, \text{ such that } a_{ip} + a_{i'p'}, \text{ and } \phi_i(s_{ij})
Step 1: Initialize <math>t_i := m_i, 1 \le i \le n$.

Step 2:

while
$$\exists (s_{ij}, s_{i'j'}) \in E$$
 s.t. $j \ge t_i$ and $j' \le t_i'$

$$\frac{do}{do} \ t_i' := t_i' - 1 \ \underline{od}$$

$$\underline{Step 3}: \ S^* := \{s_{lt_1}, \dots, s_{nt_n}\}.$$

We now prove the correctness and optimality of Algorithm 1 through a sequence of Lemmata.

LEMMA 1: Algorithm 1 always terminates.

<u>Proof:</u> The only reason it might not, is the *while* loop of Step 2. Note, however, that each time the loop is executed, exactly one t_i , is decremented by 1. Moreover, if t_i , = 0, all edges $(s_{ij}, s_{i'j'}) \in E$ violate the *while* condition because $j' \ge 1$ (as there are no edges involving $s_{i'0}$) and hence

Note that there is a convenient abuse of notation here in that we confuse the nodes of G with the savepoints.

 $j' > t_{i'}$ Therefore, no t_i , needs to be decremented below 0 and after at most $\sum_{i=1}^{n} m_i$ from the while loop will terminate.

LEMMA 2: A set of savepoints $S = \{s_{1t_1}, \dots, s_{nt_n}\}$ is consistent iff no edge of G satisfies the <u>while</u> condition in Step 2 of Algorithm 1.

<u>Proof</u>: [only if] Suppose S is consistent, yet there is an edge $(s_{ij}, s_{i'j'}) \in E$ such that $j \ge t_i$ and $j' \le t_i$. By construction of G, we have that there exist $a_{ip}, a_{i'p'}$ such that $a_{ip} \to a_{i'p'}$ and $\phi_i(s_{ij}) and <math>\phi_i, (s_{i',j'-1}) < p' \le \phi_i, (s_{i'j'})$. But then $\phi_i(s_{it_i}) \le \phi_i(s_{ij}) < p$ and $p' \le \phi_i, (s_{i'j'}) \le \phi_i, (s_{i't_i})$, violating the consistency condition.

[if] Suppose S is not consistent. That is, there exist a_{ip} and $a_{i'p'}$ such that $a_{ip} + a_{i'p}$, and (1) $p > \phi_i(s_{it_i})$ and (2) $p' \le \phi_i(s_{i't_i})$. Let j, j' be such that (3) $\phi_i(s_{ij}) and (4) <math>\phi_i(s_{i',j'-1}) < p' \le \phi_i(s_{i',j'})$. By the construction of G it follows that $(s_{ij}, s_{i'j},) \in E$. From (1), (3) it follows that $j \ge t_i$ and from (2), (4) that $j' \le t_i$. Thus $(s_{ij}, s_{i'j},)$ satisfies the condition of the while loop.

Lemma 2 implies the following

COROLLARY 1: When Algorithm 1 terminates, S* is a consistent set.

LEMMA 3: There is a unique consistent set that minimizes all reasonable functions.

 $\begin{array}{lll} & \underline{\text{Proof}}\colon \text{ Let } S=\{s_{1t_1},\ldots,s_{nt_n}\} & \text{and } S'=\{s_{1r_1},\ldots,s_{nr_n}\} & \text{be two distinct} \\ & \text{consistent sets both minimizing some reasonable function } f. & \text{That is,} \\ & f(s_{1t_1},\ldots,s_{nt_n}) = f(s_{1r_1},\ldots,s_{nr_n}). & \text{We claim that } \tilde{S}=\{s_{1q_1},\ldots,s_{nq_n}\}, \\ & \text{where } q_i = \max(t_i,r_i) & \text{is also a consistent set. For if not, by Lemma 2,} \\ & \text{there would be an edge } (s_{ij},s_{i'j'}) \in E & \text{such that } j \geqslant q_i & \text{and } j' \leqslant q_i, . \end{array}$

There are four cases to be considered: (1) if $q_i = t_i$ and $q_i = t_i$, then S is not a consistent set; (2) if $q_i = r_i$ and $q_i = r_i$, then S' is not a consistent set; (3) if $q_i = t_i$ and $q_i = r_i$, then S' is not a consistent set; and (4) if $q_i = r_i$ and $q_i = t_i$, then S is not a consistent set. All cases contradict the assumption that S and S' are consistent. Hence S is also consistent. By the definition of reasonable functions, it cannot be that $s_{it_i} < s_{ir_i}$ or that $s_{it_i} > s_{ir_i}$ for all i, for otherwise it could not be that both S and S' minimize a reasonable f. Hence $\bar{\mathsf{S}}$ is different from both S and S', and by the definition of reasonable functions, $f(s_{lq_1}, \dots, s_{nq_n}) < f(s_{lt_1}, \dots, s_{nt_n}) = f(s_{lr_1}, \dots, s_{nr_n}),$ contradicting the assumption that S and S' both minimize f. We have shown that for any given reasonable cost function f there exists a unique optimal consistent set of savepoints. It is now easy to show that the same set is optimal for all reasonable cost functions. Let f, f' be two such functions, and suppose that the respective optimal consistent sets of savepoints are S, S'. Unless $t_i = r_i$ for $1 \le i \le n$, we would have that f and f' yield a smaller cost on S (defined as above) than on the optimal S and S' respectively, a contradiction.

Lemma 3 justifies our quest for *the* optimal consistent set of savepoints (as opposed to *some* optimal such set).

Let $S = \{s_{1t_1}, \dots, s_{nt_n}\}$, $S' = \{s_{1r_1}, \dots, s_{nr_n}\}$ be sets of savepoints, one per transaction. S and S' are not necessarily consistent. We say that S transforms in one step to S', written $S \vdash S'$, iff there is an edge $(s_{ij}, s_{ij}) \in E$ such that $j \ge t_i$ and $j_0 \le t_i$, and $r_i = t_i$ for all $i \ne i_0$. Note that, according to this definition, Step 2 of Algorithm 1 repetatively transforms a set of savepoints until it cannot transform it any

more. We say that S transforms to S' iff $S \vdash^+ S'$, where \vdash^+ is the transitive closure of \vdash .

LEMMA 4: If $s_0 \vdash s_1$, $s_0 \vdash s_1$, $s_1 \vdash s$ and s is consistent, then $s_1 \vdash s$.

Proof (sketch): Recall that by the definition of $S \vdash S'$, the savepoints of S and S' are identical except those of one transaction. Let t(S,S') denote the index of that transaction—i.e. $t(S,S')=i_0$, where S,S' are as in the definition of "~" given previously. Now let $S_0 \vdash S_1 \vdash S_2 \vdash \ldots$ $S_k = S$. We also have that $S_0 \vdash S_1'$. We shall define inductively S_2',S_3',\ldots,S_k' such that $S_1' \vdash S_{1+1}'$ for $1 \le i \le k$ and $S_k' = S_k = S$. This is done as follows. Let m be the minimum index such that $t(S_{m-1},S_m)=t(S_0,S_1')$. Such an m exists, for otherwise S would not be consistent. Having defined S_1',\ldots,S_j' for some $j \le m$ we define S_{j+1}' by choosing $t(S_j',S_{j+1}')=t(S_{j-1},S_j)$. It is easy (but tedious) to verify that this can be done legally, that $S_j' \vdash S_{j+1}'$ for $1 \le j \le m$ and that $S_m' = S_m$. We may then define $S_j' = S_j$ for $m \le j \le k$, thus getting $S_0 - S_1' \vdash \ldots \vdash S_k'$, with $S_k' = S_k = S$. Hence, $S_i' \vdash S_j$ as wanted.

Lemma 4 implies that Algorithm 1, which as stated is non-deterministic, will always terminate with the same consistent set of savepoints S*, no matter in what order the edges of E are considered in Step 2.

We are now in a position to state and prove

THEOREM 1: Algorithm 1 computes the optimal consistent set of savepoints

with respect to any reasonable cost function.

Proof: Let $S^* = \{s_{1t_1}, \dots, s_{nt_n}\}$ be the set of savepoints computed by Algorithm 1, and let $S_m = \{s_{1t_1}, \dots, s_{nr_n}\}$ be the set minimizing any reasonable cost function. S_m is well-defined by Lemma 3. By that same

lemma, $r_i > t_i$ for $1 \le i \le n$. Hence, if $S^* \ne S_m$ there is non-empty set $B = \{k: 1 \le k \le n \text{ and } t_k \le r_k\}$. Consider some execution of Algorithm 1 computing S* and let $i_0 \in B$ be the element of B such that at some stage of the execution t_i was set equal to r_i in Step 2, while for all $k \in B - \{i_0\}$, $t_k > r_k$. We claim that it is possible to select edges satisfying the while condition of Step 2 so that t_i is not decremented before for all $k \in B - \{i_0\}$, $t_k = r_k$. For, if this is not the case then there exists some point in the execution of Algorithm 1 such that $t_0 = r_0$, $t_k > r_k$ for all $k \in B - \{i_0\}$, and the only edge(s) satisfying the while condition of Step 2 is (are) of the form (s_{pq}, s_{i_0q}) for some p, q, q'. Note that at this point $t_i \ge r_i$ for all $1 \le i \le n$. Then we would have that (at that point of the execution) $q \ge t_p$ and $q' \le t_i$. But we also have that $t_i = r_i$ and $t_p \ge r_i$. Therefore, $q \ge r_p$ and $q' \le r_i$ and hence S_m is not consistent by Lemma 2 (since the edge (s_{pq}, s_{i_0q}) satisfies the *while* condition for $S = S_m$). This contradicts the choice of S_{m} . Therefore, we may in fact choose edges as claimed above. But then either the algorithm will terminate before $t_i = r_i$ for all $i \in B - \{i_0\}$ --and then it would be the case that there exists some k such that $t_k > r_k$, contradicting the choice of s_m and Lemma 3--or, at some stage $t_i = r_i$ for all $i \in B$. By the definition of B, $t_i = r_i$ for all $1 \le i \le n$. Hence $S^* = S_m$, as wanted.

4. SOME REMARKS

The interdependency of operations issued by different transactions sometimes causes a transaction to restart due to a failure of some other transaction. This "domino effect" is known as cascading restart, and is usually considered to be undesirable. It is particularly serious if the

transactions that might have to be restarted are committed, as this defeats the very purpose of commitment. The usual remedy is to adopt a sufficiently conservative scheduling policy that avoids cascading restarts. If such a strategy is used the optimization problem treated in this paper is not an issue.

For example, in System-R where this savepoint feature has been implemented it is always possible to roll each transaction back to its last savepoint: this is sufficient to guarantee consistency and is obviously optimal. The reason this works is that System-R requires that all of a transaction's locks be held until that transaction is committed. Therefore, there are no dependencies between uncommitted transactions; the graph G constructed by Algorithm 1 would contain no edges; and the initial choice of savepoints $\{s_{lm}, \dots, s_{nm}\}$ would be consistent.

The optimization problem treated in this paper becomes non-trivial in environments where either cascading restarts are not an issue or methods other than restrictive schedulers are employed to avoid them. An example of such an alternative strategy would be to delay commitment. The benefit of such other strategies is that less restrictive schedulers allow greater parallelism and, therefore presumably better response time and resource utilization. We know of no study, however, which provides quantitative evidence supporting either strategy. This remains a major research area.

5. ACKNOWLEDGMENTS

I would like to express my gratitude to my advisor, Professor Phil Bernstein, and Professor Christos Papadimitriou for their support, encouragement and constructive criticisms of this work.

REFERENCES

- [A 76] Astrahan et al., "System-R, a relational approach to database management", ACM Transactions on Database Systems, 1:2, 1976, pp. 97-137.
- [F 81] Fussel, D. et al., "Deadlock removal using partial rollback in data-base systems", Proceedings of the 1981 ACM-SIGMOD Conference, pp. 65-73.
- [G 78] Gray, J., "Notes on database operating systems", Operating

 Systems: An Advanced Course, Springer-Verlag Not on Computer

 Science, 1979.
- [H 81] Hadzilacos, V., "Crash recovery in centralized database systems", unpublished manuscript, 1981.
- [Li 79] Lindsay, B.G. et al., "Notes on distributed database systems",

 IBM Research Report, 1979.
- [Lo 77] Lorie, R., "Physical integrity in a large segmented database", ACM

 Transactions on Database Systems, 2:1, 1977, pp. 91-104.
- [R 75] Rappaport, R.L., "File structure design to facilitate on-line instantaneous updating", Proceedings of the 1975 ACM-SIGMOD Conference, pp. 1-14.

LOGO COLOROS COLOROS

MISSION of Rome Air Development Center

RADC plans and executes research, development, test and selected acquisition programs in support of Command, Control Communications and Intelligence $\{C^3I\}$ activities. Technical and engineering support within areas of technical competence is provided to ESP Program Offices $\{PRS\}$ and other ESD elements. The principal technical mission areas are communications, electromagnetic guidance and control, surveillance of ground and aerospace objects, intelligence data collection and handling, information system technology, ionospheric propagation, solid state sciences, microwave physics and electronic reliability, maintainability and compatibility.

LANCARCARCARCARCARCARCARCARCARCARCARCAR

EILINED)

ALCAI

DING